Distributed Mutual Exclusion

Distributed Computing

CS-7 Second Semester

Lesson 7

Introduction to Distributed Mutual Exclusion

In a distributed system, several processes running on different machines may need access to a single shared resource. Since the resource cannot be safely used by more than one process at the same time, access must be coordinated so that only one enters its critical section at any given moment. This coordination is achieved through distributed mutual exclusion algorithms.

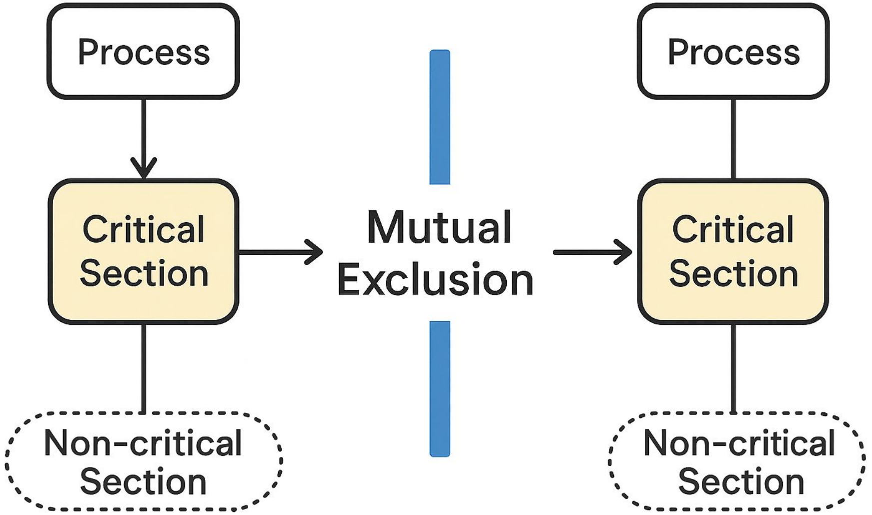
Such coordination becomes necessary because race conditions may arise if two processes update or read shared data simultaneously. Mutual exclusion ensures that shared objects such as files, buffers, or shared memory regions are accessed in a controlled sequence. Distributed systems rely on message passing to enforce this exclusivity, since no central memory or clock exists.

Deadlocks may occur when processes wait indefinitely for one another's resources. Because distributed deadlocks are harder to detect due to lack of global visibility, systematic detection and resolution mechanisms are required.

Mutual Exclusion: Fundamental Idea

Mutual exclusion represents the guarantee that no two concurrent processes execute their critical sections simultaneously. This property ensures correctness in operations involving shared data. It also prevents overlapping modifications or inconsistent reads.

A **critical section** refers to the part of a program where shared resources are accessed. Since multiple processes may reach these code segments independently, coordination is required so that only one process is active within these segments at a time.

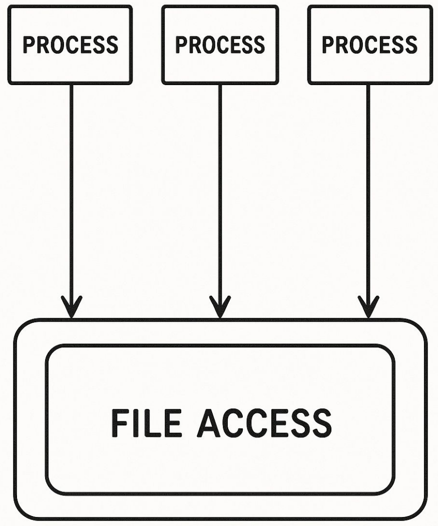


Critical Section (CS): Definition and Behaviour

A **critical section** forms the part of a multiprocess program that must not run concurrently across different processes or threads. It normally includes operations that interact with shared structures or devices. Exclusive access must be ensured to avoid interference between concurrent reads and writes.

A critical section may span multiple discontiguous parts of a program. For instance, reading a file in one part of a program and modifying it in another still requires mutual exclusion because the two actions interact with the same resource.

A thread or process attempting to enter its critical section is required to wait if another process is already inside. The critical section is expected to complete in finite time so that waiting processes are eventually served. The management of entry and exit into this section forms the core of synchronisation.



Processes Attempting to Enter a Shared File Access Routine

Preliminaries for Distributed Mutual Exclusion

Before studying specific algorithms, several foundational aspects must be considered:

System Model

A distributed system consists of multiple processes that do not share memory and instead communicate using message passing. There is no global clock, so ordering of events must be inferred using logical mechanisms.

Requirements of Mutual Exclusion Algorithms

Any distributed mutual exclusion (DME) algorithm must satisfy several essential properties:

1. **Safety**: At most one process is inside the critical section at any time.
2. **Liveness**: Every requesting process is eventually allowed to enter.
3. **Fairness**: Requests should be granted without indefinite postponement or starvation.

These requirements ensure correctness, progress, and predictable behaviour.

Performance Metrics

Performance of DME algorithms is commonly evaluated through metrics such as:

- The number of messages exchanged per critical-section entry.
- The waiting time before a process enters the critical section.

These metrics help compare assertion-based, token-based, and quorum-based approaches.

Approaches to Mutual Exclusion in Distributed Systems

There are three broad design strategies for mutual exclusion each with different trade-offs:

Token-based Approach

A unique token circulates among processes. Only the process holding the token may enter the critical section. Mutual exclusion is ensured implicitly because the token is singular. Token loss and token regeneration strategies are concerns in this category.

Non-token-based Approach

Processes exchange request and reply messages to determine order of entry. These approaches usually rely on logical clocks and message timestamps. Algorithms such as **Lamport's** and **Ricart-Agrawala** fall under this category.

Quorum-based Approach

Processes contact only a subset of nodes (a quorum) to obtain permission. Intersection of quorums ensures that mutual exclusion is preserved while reducing message overhead. **Maekawa's algorithm** uses this structure.

System Model

Structure of the Distributed System

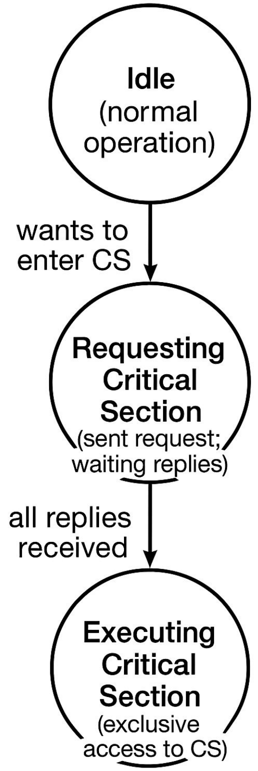
A distributed mutual exclusion environment is formed by several sites, represented as S1, S2 up to SN. Each site hosts a process, written as *pi*, and these processes communicate through an asynchronous communication network. Since no shared memory exists between sites, all coordination for entering the critical section is carried out purely through message exchanges.

When a process intends to enter its critical section, it sends REQUEST messages either to every other site or only to those sites relevant to a particular algorithm. Entry to the critical section is permitted only after appropriate replies are received. While the process waits, it is not allowed to issue another request for the same critical section, since only one outstanding request is permitted at a time.

Possible States of a Site

A site may be in one of three basic states at any moment:

- **Requesting the Critical Section**
  In this state, a process has already issued a request and is waiting for replies. It is blocked and cannot initiate another request for the critical section. This ensures a single, well-defined request per process.
- **Executing the Critical Section**
  When the site is inside its critical section, it has exclusive access to the shared resource. No other process may execute its own critical section simultaneously.
- **Idle**
  In the idle state, the site is neither requesting the critical section not executing it. The process is simply performing regular operations outside the critical region.


Idle (normal operation)
wants to enter CS
Requesting Critical Section (sent request; waiting replies)
all replies received
Executing Critical Section (exclusive access to CS)

Additional State in Token-Based Algorithms

In token-based mutual exclusion algorithms, an extra condition arises. A site may hold the token even when it is not inside the critical section. When the token is present at a site and the site is not executing the critical section, the site is considered to be in the **idle token state**.

This state highlights that holding the token does not automatically imply entry to the critical section; the site may hold the token while continuing with other work outside the critical region.

At any moment, a site may have several pending requests from other sites. These requests are queued and processed one by one so that fairness and order are preserved.

Requirements of Mutual Exclusion Algorithms

Safety Property

The safety requirement ensures that at any instant, **only one process** is allowed to execute its critical section. The purpose of this condition is to prevent conflicting operations on shared resources. Since multiple distributed processes may request the same critical section, the algorithm must guarantee that simultaneous entry never occurs. This requirement forms the foundation of correctness in distributed mutual exclusion.

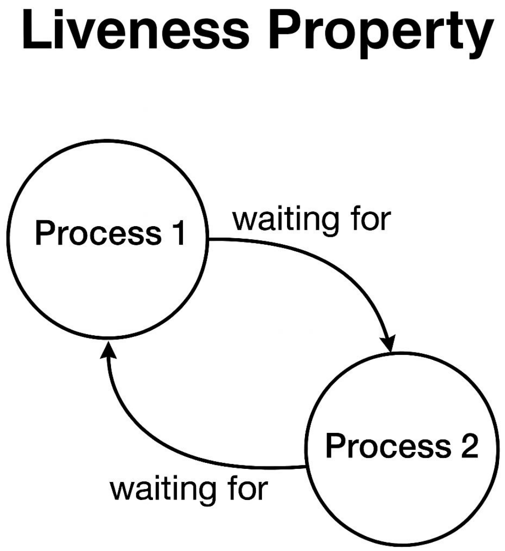
A simple conceptual example is two distributed processes attempting to update the same record. If both were allowed in the critical section at once, inconsistent or corrupted data could result. The safety property prevents such situations.

Liveness Property

Liveness addresses the guarantee that the **system continues to make progress**. It ensures that no deadlocks or starvation occur while processes compete for the critical section.

Deadlock is avoided by ensuring that sites do not wait endlessly for messages that never arrive. Starvation is prevented by ensuring that no site is forced to wait indefinitely while others repeatedly gain access to the critical section.

The idea captured in this property is that every legitimate request will eventually be served. A site wishing to execute its critical section is assured that the opportunity will arise within a finite time, provided it has made the necessary request.



Fairness Property

Fairness ensures that each process gets a **fair chance to execute** its critical section. In distributed mutual exclusion algorithms, this property is usually implemented by **respecting the order** in which requests arrive.

The sequence is typically determined by a logical time-stamping mechanism so that earlier requests are not overtaken by later ones. This prevents scenarios where one site repeatedly gains access while others are continuously postponed. The principle preserved here is that request order should reflect system arrival order as accurately as possible.

Performance Metrics for Distributed Mutual Exclusion

Message Complexity

Message complexity refers to the number of messages exchanged by a site for one execution of its critical section. Since distributed algorithms rely on message passing for coordination, this metric helps indicate how costly it is to obtain mutual exclusion. Algorithms differ significantly in how many request, reply, or token-related messages they require. A **lower message count generally implies better scalability**.

Synchronization Delay

Once a process leaves the critical section, another process may be waiting to enter. Synchronization delay measures the **time needed between one site exiting the critical section and the next site entering it**.

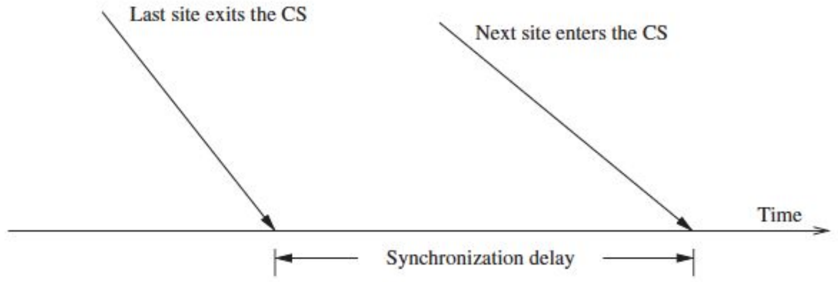
This delay typically involves one or more message exchanges necessary to transfer permission or token ownership, depending on the algorithm. Minimising synchronization delay is important because it influences how quickly the system can rotate access among processes.

Performance Metrics for Distributed Mutual Exclusion

Synchronization Delay

Once a process leaves the critical section, another process may be waiting to enter. Synchronization delay measures the **time needed between one site exiting the critical section and the next site entering it**.

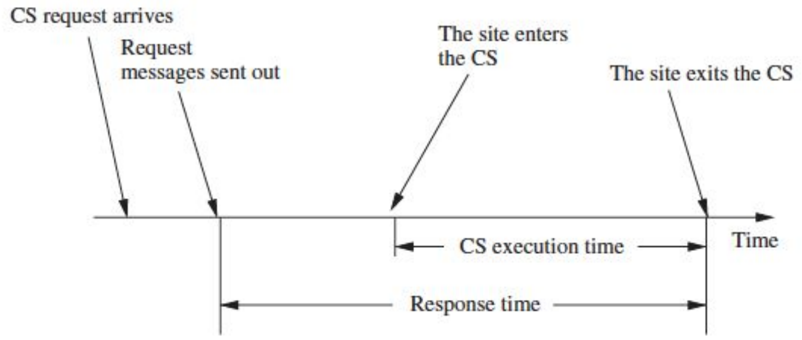
This delay typically involves one or more message exchanges necessary to transfer permission or token ownership, depending on the algorithm. Minimising synchronization delay is important because it influences how quickly the system can rotate access among processes.



Response Time

Response time refers to the period a request waits for its critical section execution to complete after its request messages have been sent out.

Important to note is that this waiting interval begins only after the messages have been issued, meaning the internal queuing delay that occurs before message transmission is not included. This metric provides a measure of how soon a process can expect to finish its critical section once its demand has been announced to other sites.



The first event shown is the arrival of the critical section request at the site. At this moment, the process has decided that it needs to enter the critical section. Shortly after this arrival, the site sends out its request messages to other sites, as required by the mutual exclusion algorithm in use.

The diagram then shows the waiting period that follows. During this time, the site cannot enter the critical section until the required permissions or replies are received. This waiting interval, starting immediately after the request messages are sent out and ending when the site actually enters the critical section, represents the **response time**. Response time measures how long the site must wait after announcing its request but does not include any time spent waiting before the request messages were transmitted.

System Throughput

System throughput captures how frequently the system can execute critical-section requests. It depends on both synchronization delay and the average time spent inside the critical section.

If **SD** denotes the **synchronization delay** and **E** represents the average critical section **execution time**, the throughput is expressed as:

$$\text{System Throughput} = 1 / (SD + E)$$

This equation reflects that higher delay or longer execution times reduce throughput, while reductions in either contribute directly to improved performance.

A small diagram showing CS time plus synchronization delay contributing to the cycle time may help visual understanding.

Distributed Mutual Exclusion: Approaches and Algorithms

Complexity of Distributed Mutual Exclusion

Distributed mutual exclusion is challenging because:

- all coordination takes place across a network where message delays are unpredictable
- and processes do not have complete knowledge of the system’s state.

Each process must make decisions with only the information available through exchanged messages. Because of this uncertainty, algorithms must be carefully designed so that correctness is maintained even when messages arrive late, arrive out of order, or are temporarily lost and later recovered.

Three broad methods are used to achieve mutual exclusion across distributed sites: token-based, non-token-based, and quorum-based approaches.

Token-Based Approach

In the token-based method, a unique token is shared among all the sites. This token is sometimes referred to as the **privilege message**. A site is permitted to enter its critical section only when it holds this token, and the site continues to hold the token until it finishes its critical section. Mutual exclusion is maintained because the token exists in exactly one copy in the system.

Different algorithms within this category vary primarily in how the system searches for the token or how the token is passed along. The main idea remains that possession of the token grants exclusive access.

Non-Token-Based Approach

This approach does not rely on a unique token. Instead, sites determine the next critical section entrant by exchanging one or more rounds of messages. A site enters its critical section only when a specific assertion, defined using its local variables, becomes true. The assertion is constructed so that it is true for only one site at any given moment, which enforces mutual exclusion.

This style of algorithm depends heavily on message ordering and evaluation of local conditions, and it avoids token-loss problems because no physical token is used.

Quorum-Based Approach

In quorum-based mutual exclusion, a site requests permission from a selected subset of sites called its quorum. The quorums are arranged so that any two quorums intersect at at least one common site. This common site ensures that when two processes try to enter the critical section at the same time, at least one site receives both requests and can enforce that only one request proceeds.

Because each site communicates only with the sites in its quorum, this approach can reduce message overhead while still maintaining correctness.

Imagine a distributed system where many sites (computers) may want to enter a **critical section** (CS).

We need a rule: **only one is allowed in at a time**.

The Core Idea

Instead of asking *every* site for permission (as in Ricart-Agrawala) or relying on a single coordinator, quorum-based mutual exclusion says:

> “Each site talks only to a **small fixed group of sites** (its *quorum*). But the quorums are arranged so cleverly that **any two quorums always overlap in at least one site**.”

That overlapping member is the key to intuition.

Think of each quorum as a small committee.

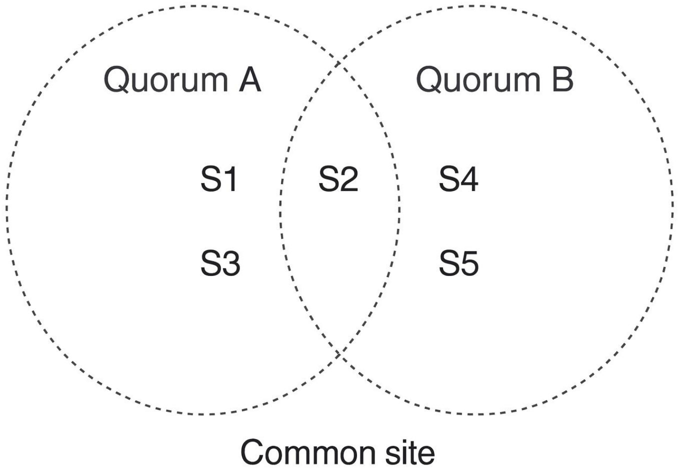
When two different sites want to enter the CS at the same time:

- Site A asks its committee (quorum A).
- Site B asks its committee (quorum B).

The **intersection site**, the one that belongs to *both* committees, will *receive both requests*.

Because the intersection site can only approve **one request at a time**, it automatically prevents both processes from entering the CS simultaneously.

That one overlapping member acts like a “referee” shared by both groups.



Why does this reduce messages?

Each site contacts only the few members in its quorum.
It does **not** need to broadcast to every site in the system.

If your quorum size is $\sqrt{N}$ in a system with N sites, the cost becomes:

- **$O(\sqrt{N})$** messages instead of **$O(N)$** in fully distributed algorithms.

So quorums are a mathematical trick that balance:

- Less communication
- With Guaranteed correctness.

Q.2 Explain the concept of Quorum used in Maekawa's algorithm. Why does Maekawa's algorithm use a RELEASE message?

Quorum Concept in Maekawa's Algorithm (Mutual Exclusion)

Maekawa's algorithm reduces message overhead by replacing "ask every site" with "ask only a fixed subset of sites," called a ***quorum****.*
A quorum is a ***small group of sites*** *selected for each process, but these quorums are constructed in a special way:*

Any two quorums always intersect in at least one common site.

This property is crucial. Suppose Process A contacts Quorum A and Process B contacts Quorum B at the same time. Because the two quorums intersect, there is at least ***one site that belongs to both****. That site receives* ***both requests****. Since each site can grant only one request at a time, the intersection site ensures that only one process obtains a full set of permissions. This overlapping site acts as a* ***referee****, preventing both processes from entering the critical section concurrently.*

Thus, quorum intersection is the mechanism that enforces ***global mutual exclusion*** *even though each process contacts only a small subset of sites.*

Why Maekawa Uses a RELEASE Message

A RELEASE message is mandatory in Maekawa's algorithm for two tightly connected reasons:

(a) To free the "grant" held at each quorum site

When a process is granted permission by its quorum, each site in that quorum temporarily becomes "locked" for that process (it cannot grant permission to anyone else).
After the process finishes its critical section, ***each quorum site must be explicitly informed that it can release this lock****.*
The RELEASE message does exactly this. It resets the site's state, allowing it to consider other pending requests.

Without RELEASE, those sites would continue thinking the process still holds the lock, causing ***permanent blocking****.*

(b) To maintain correctness across intersecting quorums

Because quorums overlap, a single site may be part of several processes' quorums.
If this common site does not receive a RELEASE message, it will not forward permission to any other process in any of those quorums.
This breaks the very reason quorums work: the intersection site must be free to arbitrate new conflicts.

Thus, RELEASE ensures:

- *no deadlock (sites are not left holding outdated grants)*
- *no starvation (waiting requests eventually get processed)*
- *no violation of mutual exclusion (the intersection site clears its state correctly before evaluating the next request)*

In short, the RELEASE message is essential to ***restore the quorum sites' availability****, maintain correct arbitration at the intersection sites, and guarantee progress in the system.*

Lamport's Distributed Mutual Exclusion Algorithm

Lamport’s distributed mutual exclusion algorithm is an application of his **logical clock synchronization scheme**. The core principle is that events, specifically **requests for the critical section**, are ordered using **logical clocks** rather than physical time.

The algorithm's **fairness** and correct execution stem from two key rules:

1. **Timestamp Assignment:** When a site receives a request, it updates its own logical clock and assigns a **timestamp** to the request; this reflects its logical ordering within the distributed system.
2. **Request Ordering:** Requests are always handled in the **increasing order of their timestamps**. This rule, which dictates that the request appearing earliest in logical time is chosen first, prevents later requests from **overtaking** earlier ones.

To enforce this ordering, each site maintains a **request queue** that stores pending mutual exclusion requests. These entries are **sorted by their timestamps**, ensuring that the request with the **smallest timestamp** is always at the front and becomes the only one eligible for execution.

Ricart-Agrawala Algorithm

The Ricart-Agrawala algorithm is a permission-based distributed mutual exclusion protocol that builds on Lamport's synchronization principles and optimizes message complexity. It fundamentally relies on FIFO communication channels and two message types: REQUEST for entering the critical section and REPLY for granting access.

Key Features and Steps

- Each process sends a REQUEST, timestamped by its Lamport-style logical clock, to every other process when it wants to enter its critical section.
- Receiving processes use the timestamp to compare priorities: if the incoming request's timestamp is earlier (higher priority), and the receiver is not in its own critical section, it sends a REPLY immediately. Otherwise, the REPLY is deferred until the receiver exits the critical section.
- A process can only enter its critical section after receiving a REPLY from every other process, ensuring the earliest request in logical time always proceeds first.
- Message complexity is optimal, requiring only 2(N−1) messages for each critical section entry (REQUESTs and REPLYs), significantly reducing overhead compared to earlier algorithms.

Correctness and Fairness

- Timestamps ensure strict ordering; lower (earlier) timestamps always take priority over higher ones.
- Starvation is prevented because every deferred REPLY is eventually sent when a process completes its critical section.
- Deadlock is avoided as the algorithm guarantees progress for the request with the smallest timestamp.
- The approach optimizes overhead by removing the need for explicit RELEASE messages, relying instead on deferred REPLYs after the critical section is released.

Maekawa’s Algorithm

Maekawa’s algorithm works by giving each site a predefined group of sites called its ***request set***.

- **(M1)** When a site wants to enter its critical section, it must get permission from every site in this group. The key idea is that the request sets overlap: any two sites’ request sets share at least one common site. Because of this overlap, at least one site will always see both requests if two sites request the critical section at the same time. This common site acts as the mediator, ensuring that only one request is allowed to proceed.
- **(M2)** Each site is allowed to give out permission to only one ***requester*** at a time. This means a site cannot send more than one REPLY message without first receiving a RELEASE message for the previous permission that it granted.

If a new REQUEST arrives while a site has already granted permission to someone else, the request cannot be granted immediately. Instead, it is placed in a queue. The queued request will be served later, but only after the site receives a RELEASE message from the currently permitted site. This rule maintains mutual exclusion across the entire system.

Two main conditions, M1 and M2, ensure that the algorithm works correctly. Conditions M3 and M4 are additional improvements.

- Condition M3 ensures that every site has a request set of the same size. This balances the amount of work each site performs when handling mutual exclusion requests.
- Condition M4 ensures that every site is asked for permission by the same number of other sites. This keeps the “responsibility load” evenly distributed, so no single site becomes a bottleneck.

Requesting the Critical Section

When a site **Si** wants to enter its critical section, it sends a **REQUEST(i)** message to every site in its request set **Ri**. These sites serve as a group whose approval is required before **Si** can proceed.

When a site **Sj** receives a REQUEST(i), it first checks whether it has already given permission (a **REPLY** message) to another site since the last time it received a RELEASE.

- If **Sj has not granted permission to anyone**, it is free to approve Si’s request and immediately sends **REPLY(j)** to Si.
- If **Sj has already granted permission**, it cannot approve Si’s request at that moment, so it places the request in a **queue** for later handling.

This queueing approach ensures that Sj deals with incoming requests one at a time, in the order received, and only grants permission when it is safe to do so.

Across the entire request set **Ri**, Si will be allowed to enter the critical section only after it receives a REPLY from *every* site in Ri. This guarantees mutual exclusion because all required sites have explicitly granted permission before Si enters the critical section.

Releasing the Critical Section

Once **Si** finishes executing its critical section, it must inform all the sites in its request set **Ri** that it no longer needs the permission they gave. To do this, **Si sends a RELEASE(i) message to every site in Ri**. This message tells those sites that the permission previously granted to Si is no longer in use.

When a site **Sj** receives a RELEASE(i) message from Si, it performs one of two actions, depending on its queue:

1. If Sj's queue has waiting requests

Sj now becomes free to grant permission again. It looks at the **next request in its queue**, sends a **REPLY(j)** to that site, and removes that request from the queue.

2. If Sj's queue is empty

Sj simply updates its internal state to show that it is no longer holding an outstanding REPLY. In other words, it is now ready to immediately grant permission to the next REQUEST it receives.

This behaviour ensures that each site grants permission fairly and in proper order, and that no site holds permission longer than necessary.