Distributed Mutual Exclusion

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss about the 'Concepts of Mutual Exclusion', Classical algorithms for distributed computing systems and Industry systems for Mutual Exclusion.

Need of Mutual Exclusion in Cloud

- **Bank's Servers in the Cloud:** Two customers make simultaneous deposits of 10,000 Rs. into your bank account, each from a separate ATM.
  - Both ATMs read initial amount of 1000 Rs. concurrently from the bank's cloud server
  - Both ATMs add 10,000 Rs. to this amount (locally at the ATM)
  - Both write the final amount to the server
  - **What's wrong? 11000Rs. (or 21000Rs.)**

Need of Mutual Exclusion in Cloud

- **Bank's Servers in the Cloud:** Two customers make simultaneous deposits of 10,000 Rs. into your bank account, each from a separate ATM.
  - Both ATMs read initial amount of 1000 Rs. concurrently from the bank's cloud server
  - Both ATMs add 10,000 Rs. to this amount (locally at the ATM)
  - Both write the final amount to the server
  - You lost 10,000 Rs.!
- The ATMs need ***mutually exclusive*** access to your account entry at the server
  - or, mutually exclusive access to executing the code that modifies the account entry

Some other Mutual Exclusion use

- **Distributed File systems**
  - Locking of files and directories
- **Accessing objects** in a safe and consistent way
  - Ensure at most one server has access to object at any point of time
- **Server coordination**
  - Work partitioned across servers
  - Servers coordinate using locks
- **In industry**
  - Chubby is Google's locking service
  - Many cloud stacks use Apache Zookeeper for coordination among servers

Problem Statement for Mutual Exclusion

- ***Critical Section*** Problem: Piece of code (at all processes) for which we need to ensure there is <u>at most one process</u> executing it at any point of time.
- Each process can call three functions
  - **enter()** to enter the critical section (CS)
  - **AccessResource()** to run the critical section code
  - **exit()** to exit the critical section

Bank Example

ATM1:

```
enter(S);
 // AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
// AccessResource() end
exit(S); // exit
```

ATM2:

```
enter(S);
 // AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
// AccessResource() end
exit(S); // exit
```

Approaches to Solve Mutual Exclusion

- **Single OS:**
  - If all processes are running in one OS on a machine (or VM), then
  - Semaphores, mutexes, condition variables, monitors, etc.

Approaches to Solve Mutual Exclusion (2)

- Distributed system:
  - Processes communicating by passing messages

Need to guarantee 3 properties:

- **Safety** (essential): At most one process executes in CS (Critical Section) at any time
- **Liveness** (essential): Every request for a CS is granted eventually
- **Fairness** (desirable): Requests are granted in the order they were made

Processes Sharing an OS: Semaphores

- Semaphore == an integer that can only be accessed via two special functions
- Semaphore S=1; // Max number of allowed accessors

1. wait(S) (or **P(S)** or **down(S)**):

enter()

```
while(1) { // each execution of the while loop is atomic
        if (S > 0) {
            S--;
            break;
        }
 }
```

Each while loop execution and S++ are each atomic operations – supported via hardware instructions such as compare-and-swap, test-and-set, etc.

exit() 2. **signal(S)** (or **V(S)** or **up(s)**):

```
S++; // atomic
```

Bank Example Using Semaphores

```
Semaphore S=1; // shared
ATM1:
    wait(S);
     // AccessResource()
    obtain bank amount;
    add in deposit;
    update bank amount;
    // AccessResource() end
    signal(S); // exit
```

```
Semaphore S=1; // shared
ATM2:
    wait(S);
     // AccessResource()
    obtain bank amount;
    add in deposit;
    update bank amount;
    // AccessResource() end
    signal(S); // exit
```

Next

- In a distributed system, cannot share variables like semaphores
- So how do we support mutual exclusion in a distributed system?

System Model

- Before solving any problem, specify its System Model:

  - Each pair of processes is connected by reliable channels (such as TCP).
  - Messages are eventually delivered to recipient, and in FIFO (First In First Out) order.
  - Processes do not fail.
    - Fault-tolerant variants exist in literature.

Central Solution

- Elect a central master (or leader)
  - Use one of our election algorithms!
- Master keeps
  - A **queue** of waiting requests from processes who wish to access the CS
  - A special **token** which allows its holder to access CS
- Actions of any process in group:
  - **enter()**
    - Send a request to master
    - Wait for token from master
  - **exit()**
    - Send back token to master

Central Solution

- Master Actions:
  - On receiving a request from process P*i*

    if (master has token)

    Send token to P*i*

    else

    Add P*i* to queue
  - On receiving a token from process P*i*

    if (queue is not empty)

    Dequeue head of queue (say P*j*), send that process the token

    else

    Retain token

Analysis of Central Algorithm

- Safety – at most one process in CS
  - Exactly one token
- Liveness – every request for CS granted eventually
  - With *N* processes in system, queue has at most *N* processes
  - If each process exits CS eventually and no failures, liveness guaranteed
- FIFO Ordering is guaranteed, in order of requests received at master

Performance Analysis

Efficient mutual exclusion algorithms use **fewer messages**, and make processes **wait for shorter durations** to access resources. Three metrics:

- ***Bandwidth***: the total number of messages sent in each *enter* and *exit* operation.
- ***Client delay***: the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)

  (We will prefer mostly the enter operation.)
- ***Synchronization delay***: the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting)

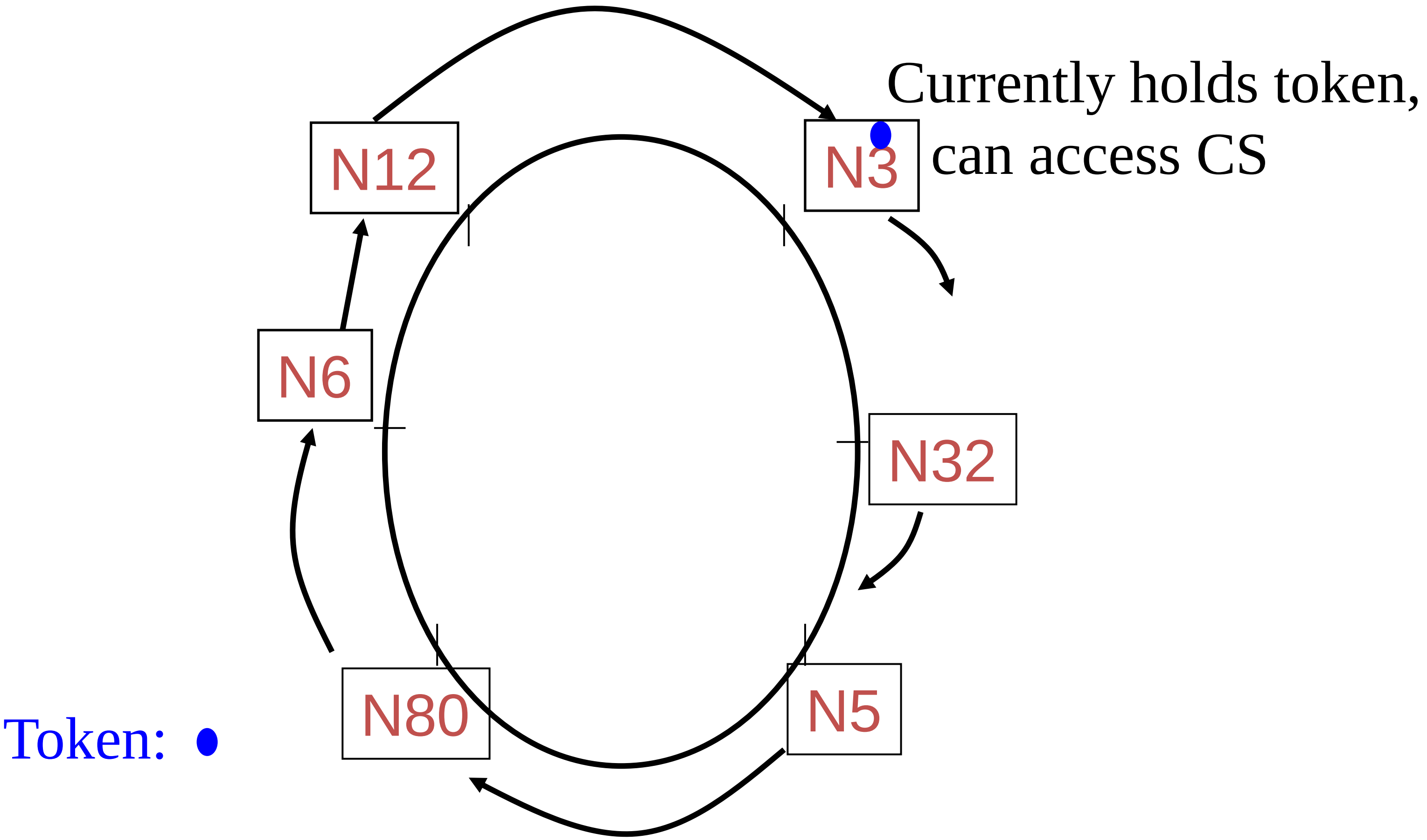
Analysis of Central Algorithm

- ***Bandwidth***: the total number of messages sent in each *enter* and *exit* operation.
  - 2 messages for enter
  - 1 message for exit
- ***Client delay***: the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)
  - 2 message latencies (request + grant)
- ***Synchronization delay***: the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting)
  - 2 message latencies (release + grant)

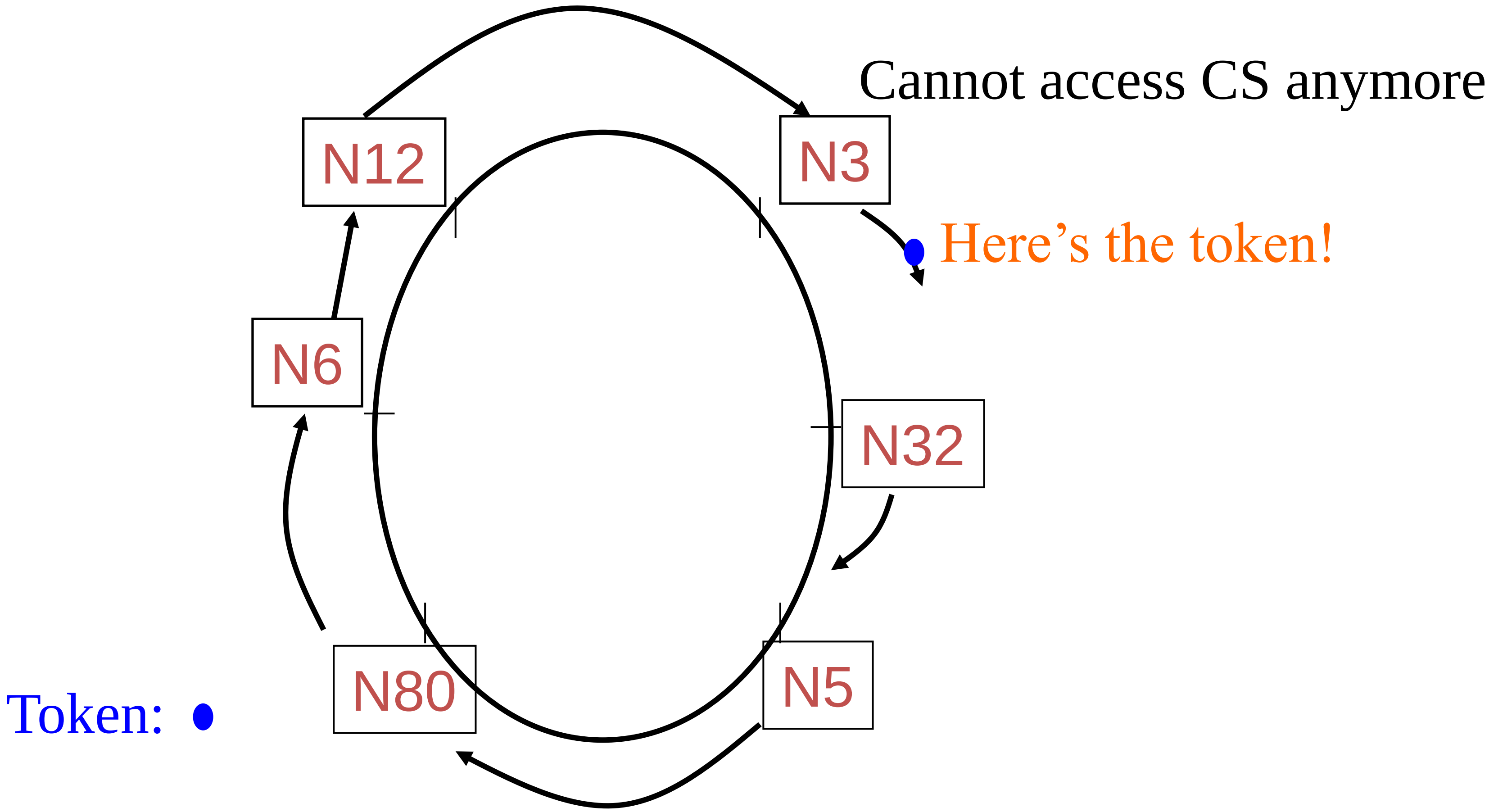
But...

- The master is the performance bottleneck and SPoF (single point of failure)

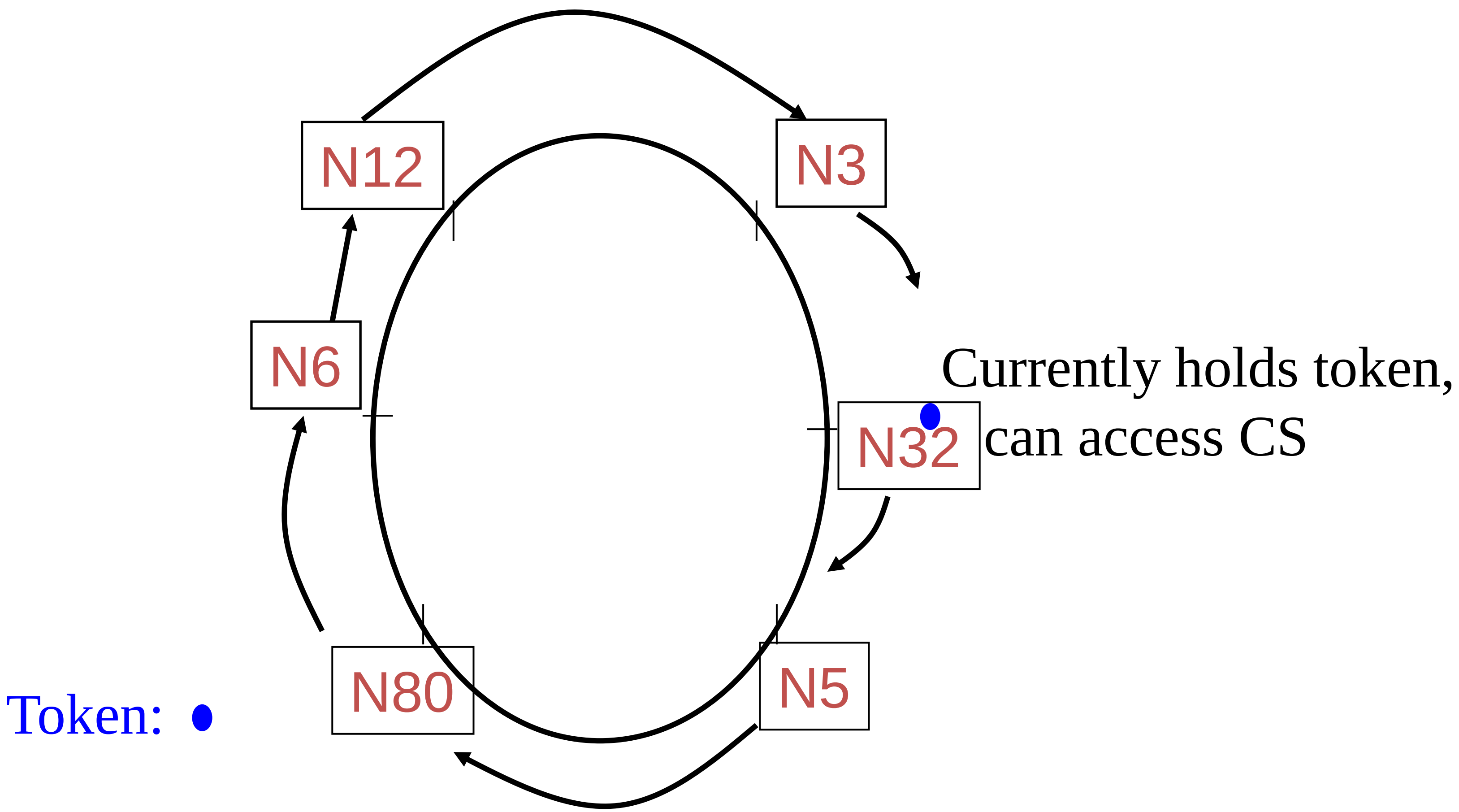
Ring-based Mutual Exclusion



Ring-based Mutual Exclusion



Ring-based Mutual Exclusion



Ring-based Mutual Exclusion

- *N* Processes organized in a virtual ring
- Each process can send message to its successor in ring
- Exactly 1 token
- enter()
  - Wait until you get token
- exit() // already have token
  - Pass on token to ring successor
- If receive token, and not currently in enter(), just pass on token to ring successor

Analysis of Ring-based Mutual Exclusion

- Safety
  - Exactly one token
- Liveness
  - Token eventually loops around ring and reaches requesting process (no failures)
- Bandwidth
  - Per enter(), 1 message by requesting process but up to *N* messages throughout system
  - 1 message sent per exit()

Analysis of Ring-Based Mutual Exclusion (2)

- Client delay: 0 to *N* message transmissions after entering enter()
  - Best case: already have token
  - Worst case: just sent token to neighbor
- Synchronization delay between one process' exit() from the CS and the next process' enter():
  - Between 1 and (*N-1*) message transmissions.
  - Best case: process in enter() is successor of process in exit()
  - Worst case: process in enter() is predecessor of process in exit()

Next

- Client/Synchronization delay to access CS still O(*N*) in Ring-Based approach.
- Can we make this faster?

System Model

- Before solving any problem, specify its System Model:
  - Each pair of processes is connected by reliable channels (such as TCP).
  - Messages are eventually delivered to recipient, and in FIFO (First In First Out) order.
  - Processes do not fail.

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

<u>**Releasing the critical section:**</u>

- 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

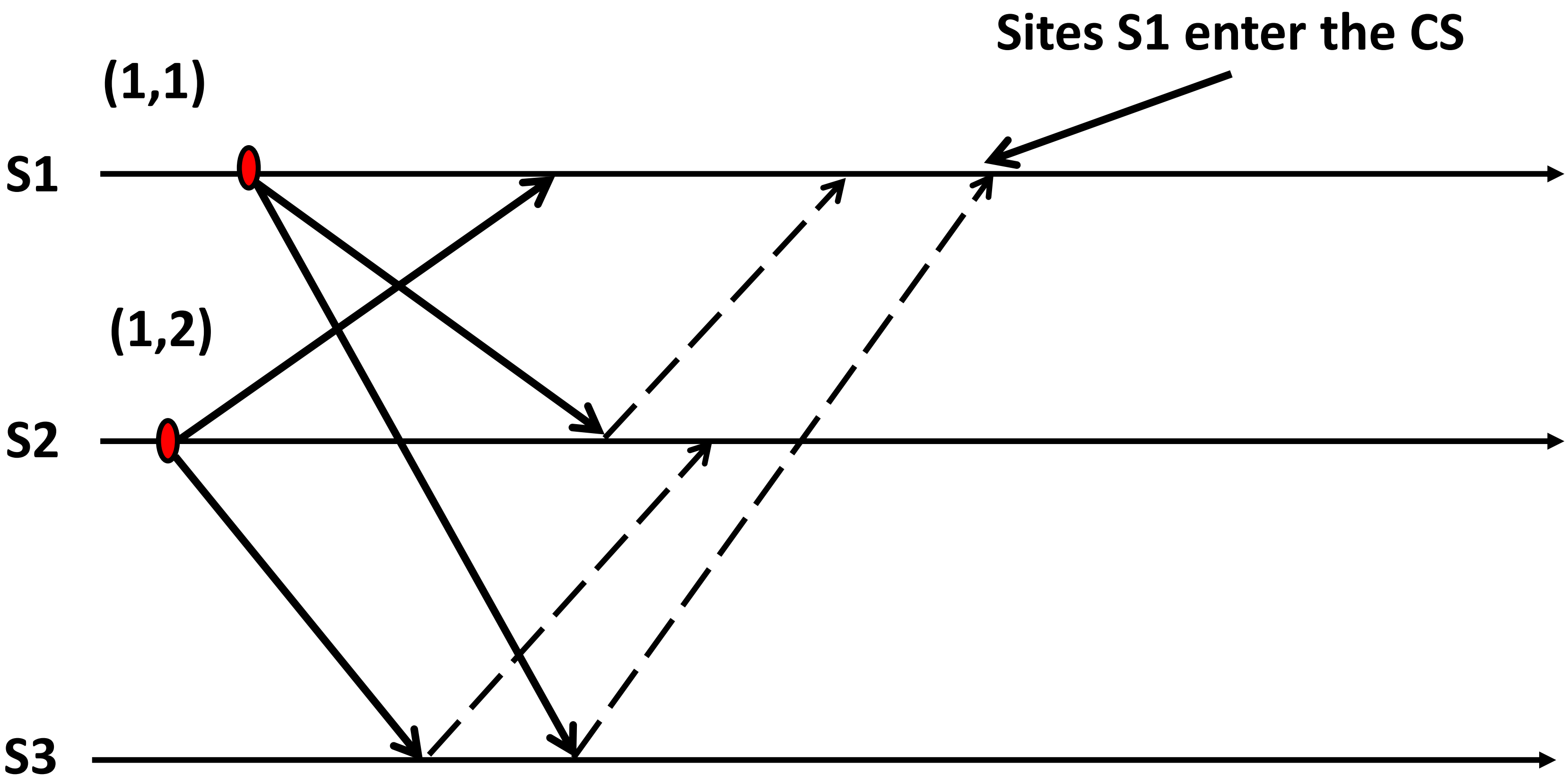
Theorem: Lamport's algorithm is fair.

Proof:

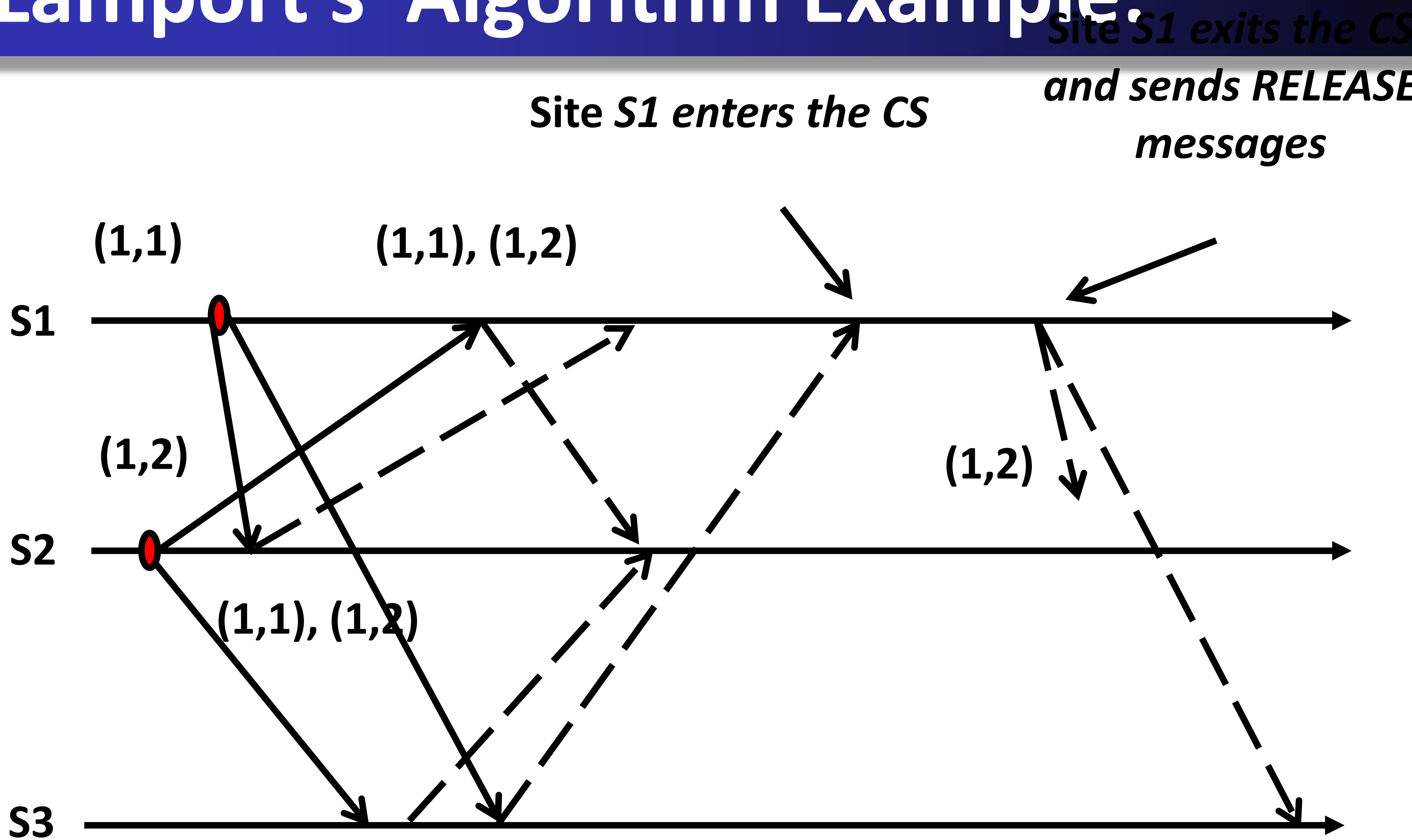
- The proof is by contradiction. Suppose a site S_i's request has a smaller timestamp than the request of another site S_j and S_j is able to execute the CS before S_i.
- For S_j to execute the CS, it has to satisfy the conditions **L1** and **L2**. This implies that at some instant in time say t, S_j 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:

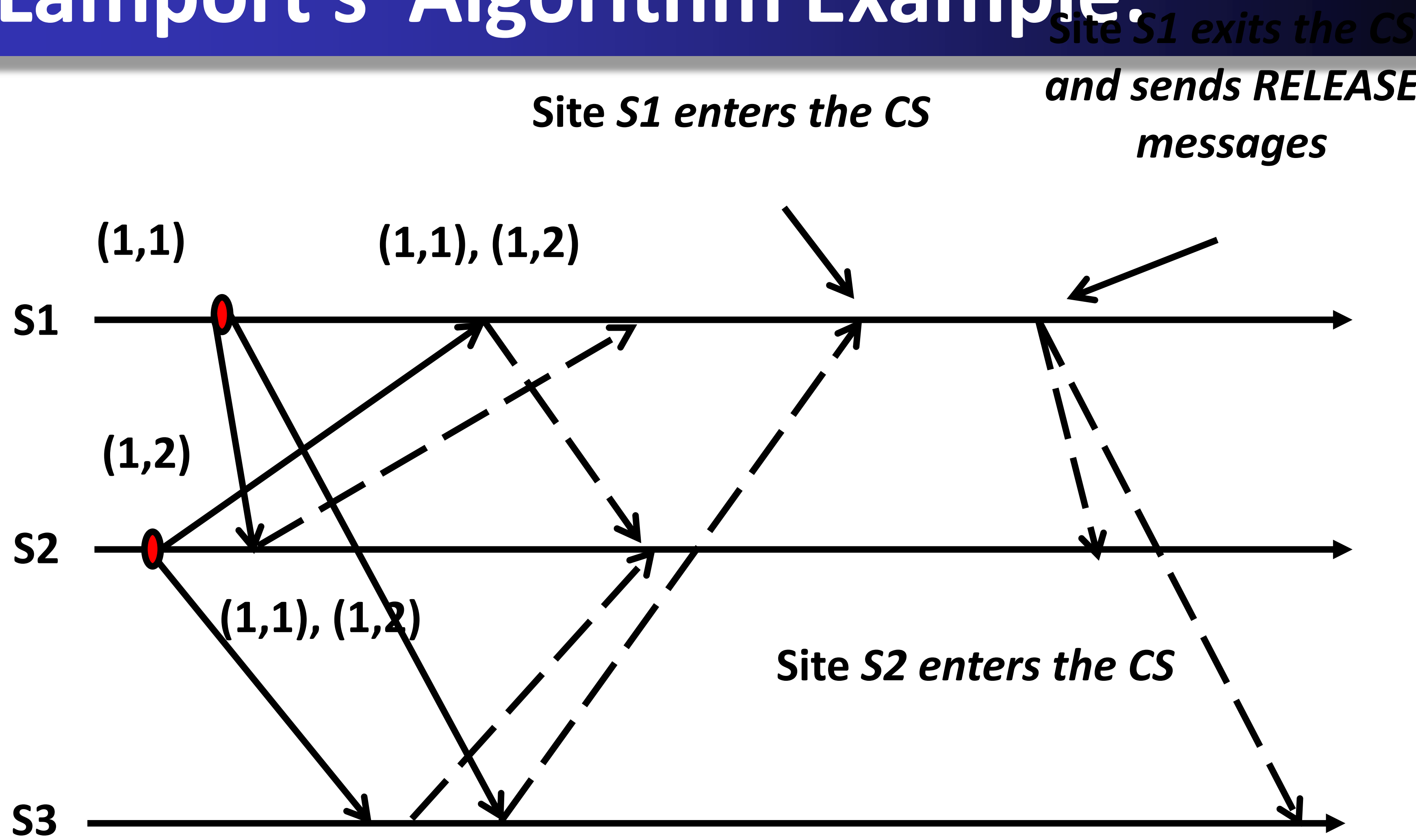
Sites S1 and S2 are Making Requests for the CS



Lamport's Algorithm Example:



Lamport's Algorithm Example:



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.

Ricart-Agrawala's Algorithm

- Classical algorithm from 1981
- Invented by Glenn Ricart (NIH) and Ashok Agrawala (U. Maryland)

- No token
- Uses the notion of causality and multicast
- Has lower waiting time to enter CS than Ring-Based approach

Key Idea: Ricart-Agrawala Algorithm

- enter() at process P*i*
  - multicast a request to all processes
    - Request: <T, P*i*>, where T = current Lamport timestamp at P*i*
  - Wait until *all* other processes have responded positively to request
- Requests are granted in order of causality
- <T, P*i*> is used lexicographically: P*i* in request <T, P*i*> is used to break ties (since Lamport timestamps are not unique for concurrent events)

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

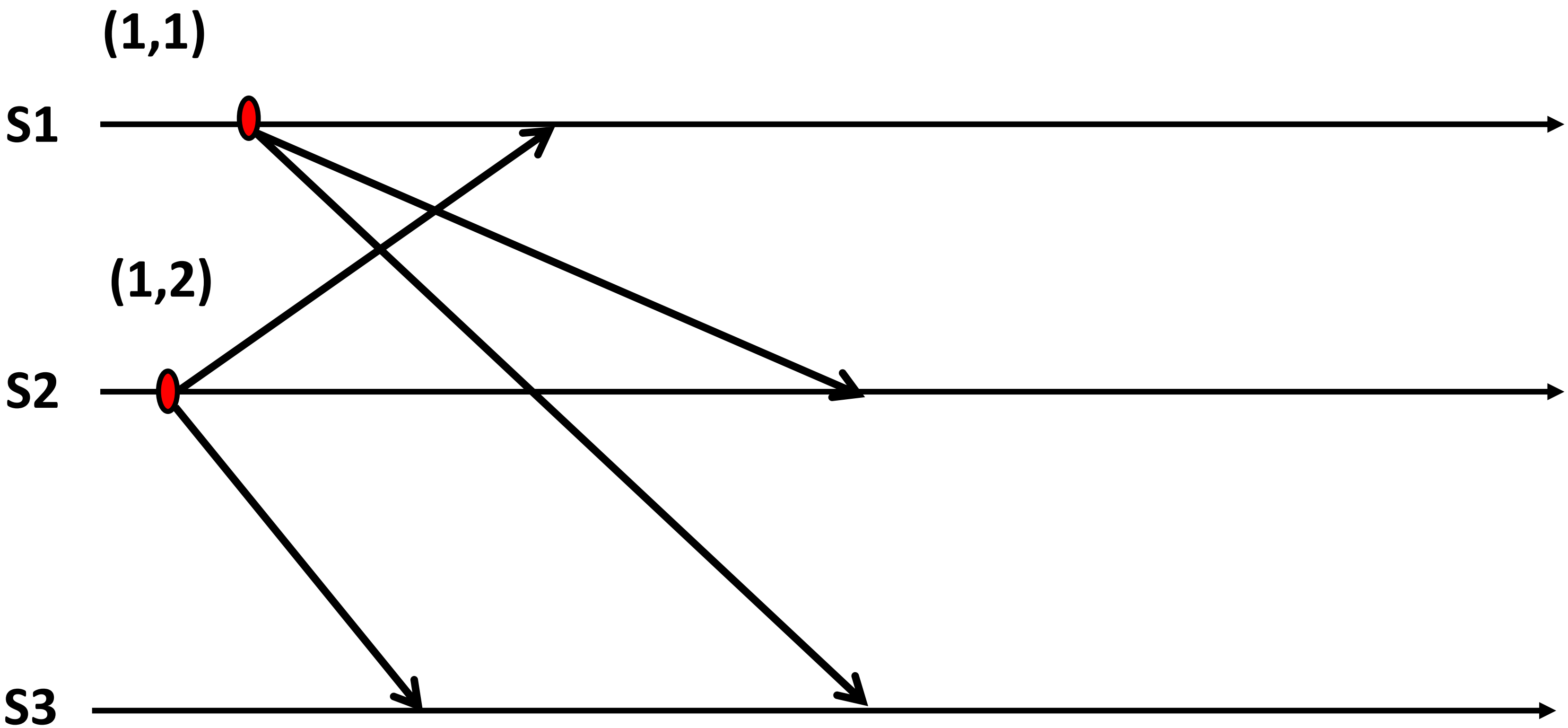
Theorem: Ricart-Agrawala algorithm achieves mutual exclusion.

Proof:

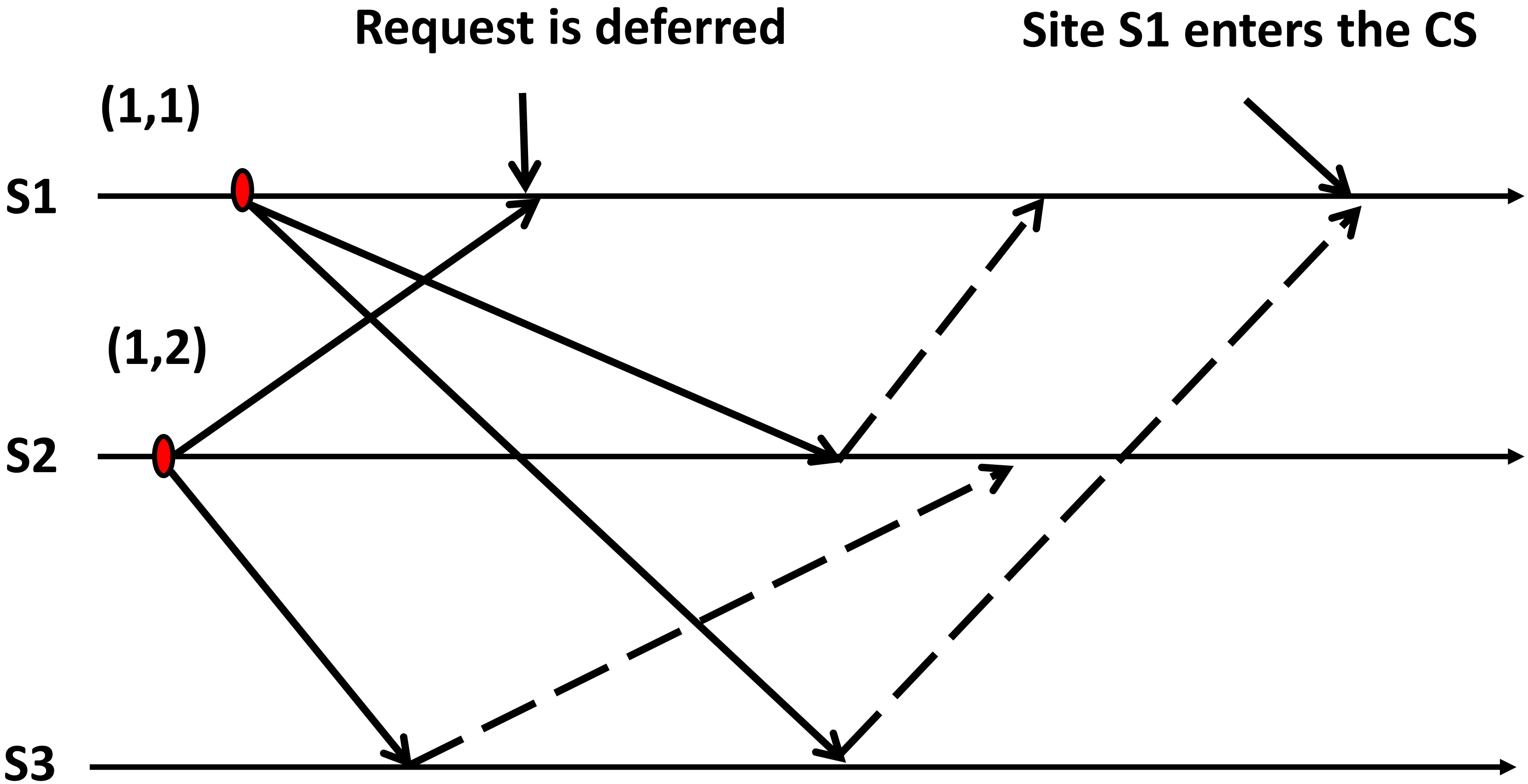
- Proof is by contradiction. Suppose two sites S_i and S_j' are executing the CS concurrently and S_i's request has higher priority than the request of S_j. Clearly, S_i received S_j's request after it has made its own request.
- Thus, S_j can concurrently execute the CS with S_i only if S_i returns 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:

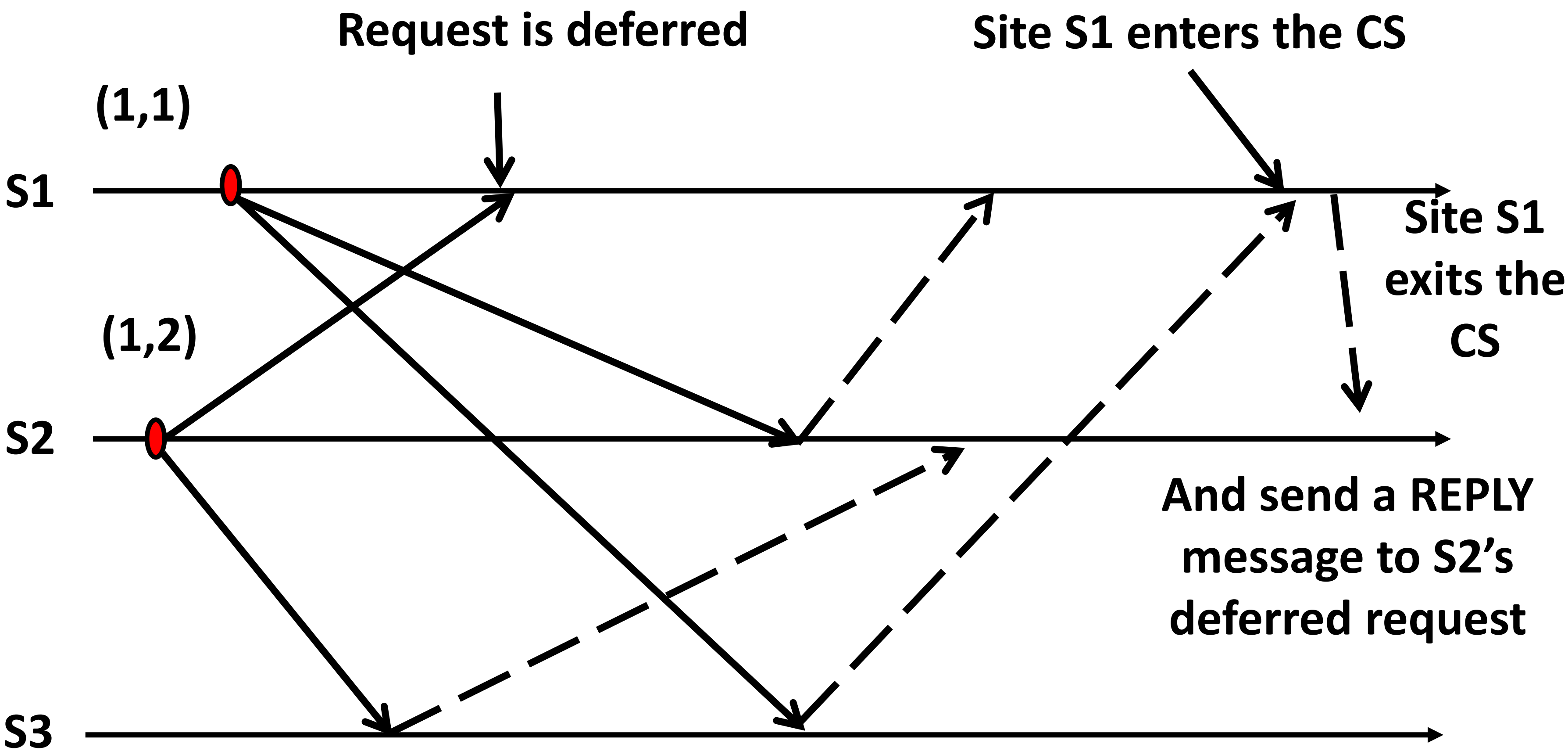
Sites S1 and S2 are Making Requests for the CS



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*** .

Comparison

- Compared to Ring-Based approach, in Ricart-Agrawala approach
  - Client/synchronization delay has now gone down to O(1)
  - But bandwidth has gone up to O(*N*)
- Can we get *both* down?

Quorum-based approach

- In the '**quorum-based approach**', each site requests permission to execute the CS from a **subset of sites** (called a **quorum**).

- The **intersection property of quorums** make sure that only one request executes the CS at any time.

Quorum-Based Mutual Exclusion Algorithms

Quorum-based mutual exclusion algorithms are different in **two** ways:

1. A site does not request permission from **all other sites**, but only from a **subset of the sites**.

   *The **request set** of sites are chosen such that*

   $\forall i\ \forall j: 1 \le i, j \le N :: R_i \cap R_j \neq \Phi.$

   Consequently, every pair of sites has a site which mediates conflicts between that pair.

2. A site can send out only **one REPLY** message at any time.

   A site can send a **REPLY** message only after it has received a **RELEASE** message for the previous **REPLY** message.

Contd...

Notion of '**Coteries**' and '**Quorums**':

A **coterie *C*** is defined as a set of sets, where each set $g \in C$ is called a **quorum.**

The following properties hold for quorums in a coterie:

- **Intersection property:** For every quorum $g, h \in C$, $g \cap h \neq \emptyset$.
  For example, sets {1,2,3}, {2,5,7} and {5,7,9} cannot be quorums in a coterie, because first and third sets **do not have a common element**.

- **Minimality property:** There should be no quorums g, h in **coterie C** such that **$g \supseteq h$ i.e g is superset of h**.
  For example, sets {1,2,3} and {1,3} cannot be quorums in a coterie because the first set is a superset of the second.

Maekawa's Algorithm

Maekawa's algorithm was **first quorum-based mutual exclusion algorithm**.

- The **request sets for sites** (i.e., **quorums**) in Maekawa's algorithm are constructed to satisfy the following conditions:

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 $|R_i| = \sqrt{N}$

Maekawa's Algorithm

- Conditions **M1** and **M2** are necessary for correctness; whereas conditions **M3** and **M4** provide other desirable features to the algorithm.
- Condition **M3** states that the size of the requests sets of all sites must be equal implying that all sites should have to do an equal amount of work to invoke mutual exclusion.
- Condition **M4** enforces that exactly the same number of sites should request permission from any site, which implies that all sites have **"equal responsibility"** in granting permission to other sites.

The Algorithm

A site S_i executes the following steps to execute the CS.

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 .

The Algorithm

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.

Correctness

Theorem: ***Maekawa's algorithm achieves mutual exclusion.***
Proof:

- Proof is by contradiction. Suppose two sites S_i and S_j are concurrently executing the CS.

- This means site S_i received a **REPLY** message from all sites in R_i and concurrently site S_j was able to receive a **REPLY** message from all sites in R_j.

- If $R_i \cap R_j = \{S_k\}$, then site S_k must have sent **REPLY** messages to both S_i and S_j concurrently, which is a contradiction.

Performance

- Since the size of a request set is $\sqrt{N}$, an execution of the CS requires $\sqrt{N}$ REQUEST, $\sqrt{N}$ REPLY, and $\sqrt{N}$ RELEASE messages, resulting in $3\sqrt{N}$ 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.

Problem of Deadlocks

- **Maekawa's algorithm can deadlock** because a site is exclusively locked by other sites and requests are not prioritized by their timestamps.

  Assume three sites **S_i, S_j, and S_k simultaneously invoke mutual exclusion**.
  - Suppose $R_i \cap R_j = \{S_{ij}\}$, $R_j \cap R_k = \{S_{jk}\}$, and $R_k \cap R_i = \{S_{ki}\}$.

Consider the following scenario:

1. S_{ij} has been locked by S_i (forcing S_j to wait at S_{ij}).
2. S_{jk} has been locked by S_j (forcing S_k to wait at S_{jk}).
3. S_{ki} has been locked by S_k (forcing S_i to wait at S_{ki}).

- This state represents a deadlock involving sites S_i, S_j, and S_k.

Handling Deadlocks

- Maekawa's algorithm handles deadlocks by requiring a site to yield a lock if **the timestamp of its request is larger than the timestamp of some other request** waiting for the same lock.

- A site **suspects a deadlock** (and initiates message exchanges to resolve it) whenever a higher priority request arrives and waits at a site because the site has sent a **REPLY message to a lower priority request**.

Message types for Handling Deadlocks

Deadlock handling requires **three types of messages**:

FAILED: A FAILED message from site S_i to site S_j indicates that S_i can not grant S_j's request because it has currently granted permission to a site with a higher priority request.

INQUIRE: An INQUIRE message from S_i to S_j indicates that S_i would like to find out from S_j if it has succeeded in locking all the sites in its request set.

YIELD: A YIELD message from site S_i to S_j indicates that S_i is returning the permission to S_j (to yield to a higher priority request at S_j).

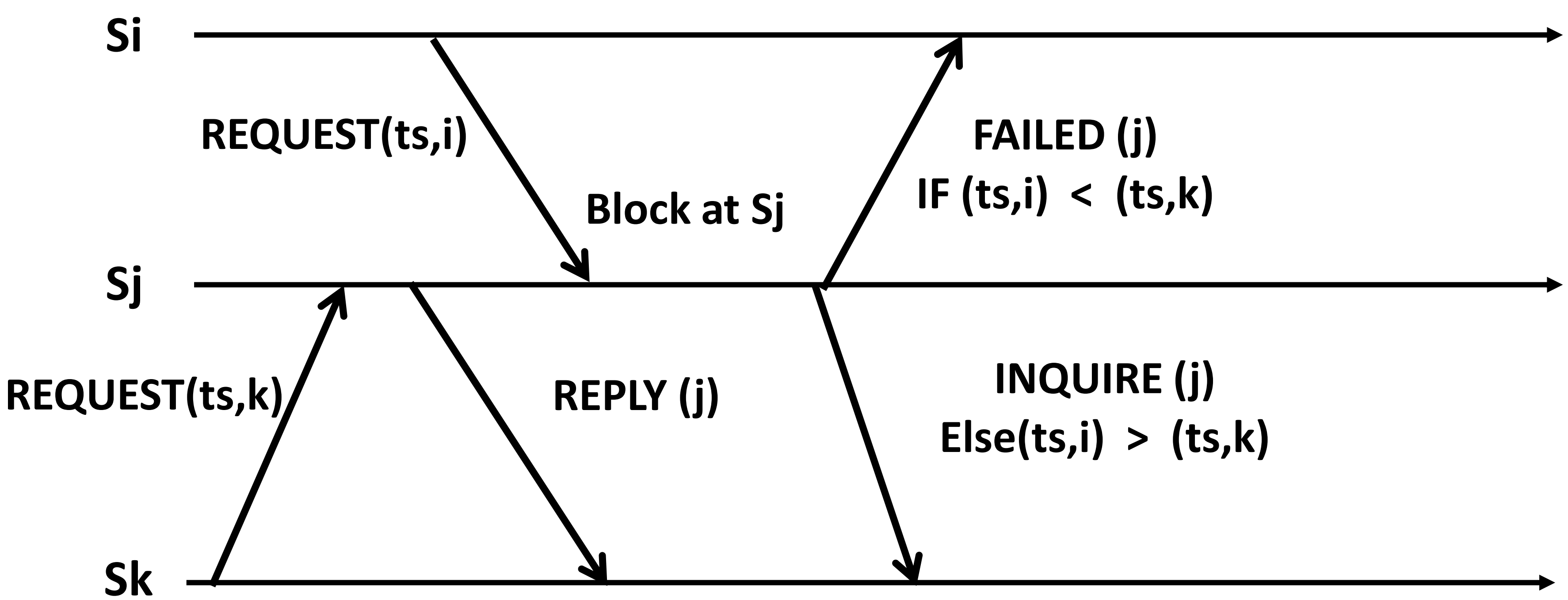
Handling Deadlocks

Maekawa's algorithm handles deadlocks as follows:

- When a **REQUEST(*ts* , *i*)** from site S_i blocks at site S_j because S_j has currently granted permission to site S_k, then S_j sends a **FAILED(*j*)** message to S_i if S_i's request has lower priority. Otherwise, S_j sends an **INQUIRE(*j*)** message to site S_k.
- In response to an **INQUIRE(*j*)** message from site S_j, site S_k sends a **YIELD(*k*)** message to S_j provided S_k has received a **FAILED** message from a site in its request set and if it sent a **YIELD** to any of these sites, but has not received a new **REPLY** from it.
- In response to a **YIELD(*k*)** message from site S_k, site S_j assumes as if it has been released by S_k, places the request of S_k at appropriate location in the request queue, and sends a **REPLY(*j*)** to the top request's site in the queue.
- **Maximum number of messages** required per CS execution in this case is 5 $\sqrt{N}$

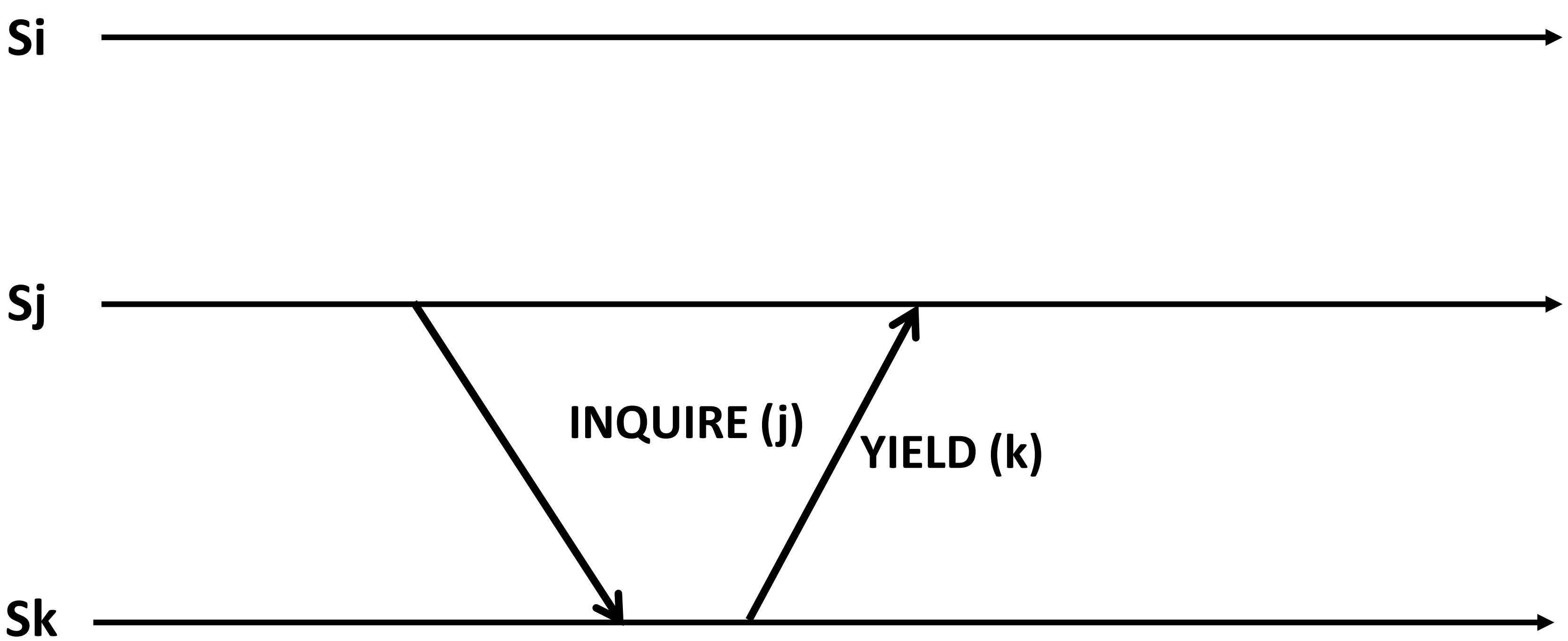
Handling Deadlocks: Case-I

When a **REQUEST(*ts* , *i*)** from site S_i blocks at site S_j because S_j has currently granted permission to site S_k , then S_j sends a **FAILED(*j*)** message to S_i if S_i's request has lower priority. Otherwise, S_j sends an **INQUIRE(*j*)** message to site S_k .



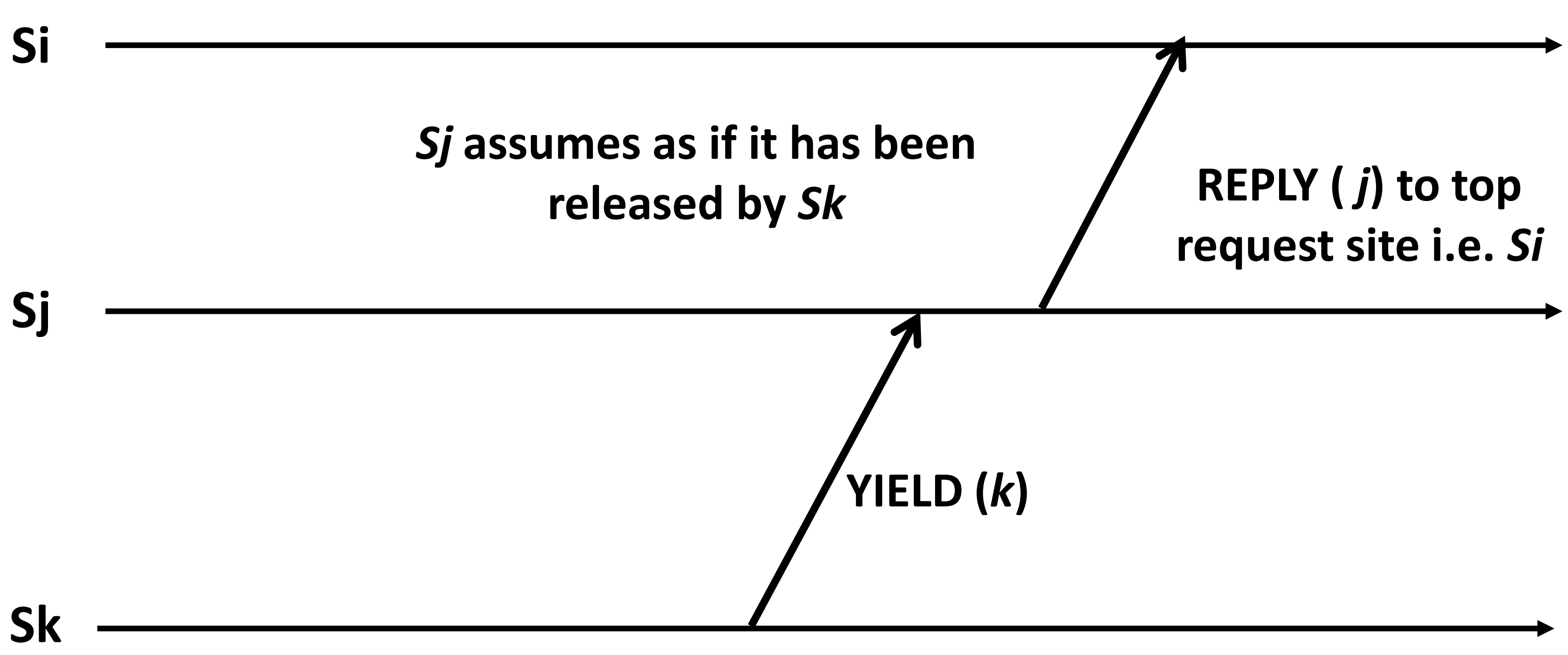
Handling Deadlocks: Case-II

In response to an **INQUIRE(*j*)** message from site S_j, site S_k sends a **YIELD(*k*)** message to S_j provided S_k has received a **FAILED** message from a site in its request set and if it sent a **YIELD** to any of these sites, but has not received a new **REPLY** from it.



Handling Deadlocks: Case-III

In response to a **YIELD(*k*)** message from site S_k , site S_j assumes as if it has been released by S_k , places the request of S_k at appropriate location in the request queue, and sends a **REPLY(*j*)** to the top request's site in the queue.



Failures?

- other ways to handle failures: Use Paxos like!

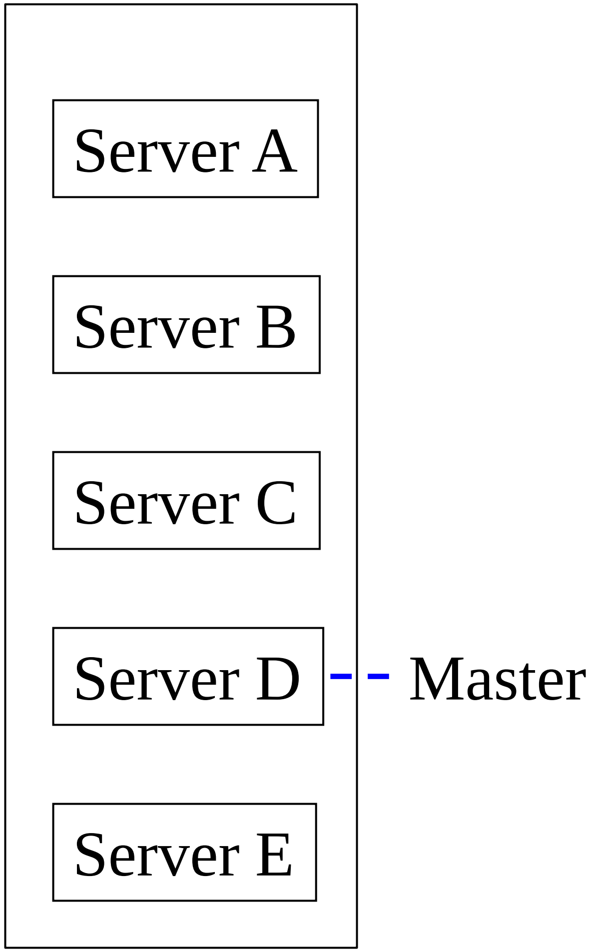
Industry Mutual Exclusion : Chubby

- Google's system for locking
- Used underneath Google's systems like BigTable, Megastore, etc.
- Chubby provides *Advisory* locks only
  - Doesn't guarantee mutual exclusion unless every client checks lock before accessing resource

Reference: *http://research.google.com/archive/chubby.html*

Chubby

- Can use not only for locking but also writing small configuration files
- Relies on Paxos-like (consensus) protocol
- Group of servers with one elected as Master
  - All servers replicate same information
- Clients send **read** requests to Master, which serves it **locally**
- Clients send **write** requests to Master, which sends it to all servers, gets **majority (quorum)** among servers, and then responds to client
- On master failure, run election protocol
- On replica failure, just replace it and have it catch up



Conclusion

- Mutual exclusion important problem in cloud computing systems
- Classical algorithms
  - Central
  - Ring-based
  - Lamport's Algorithm
  - Ricart-Agrawala
  - Maekawa
- Industry systems
  - Chubby: a coordination service
  - Similarly, Apache Zookeeper for coordination