UNIT III DISTRIBUTED MUTEX & DEADLOCK 9

Distributed mutual exclusion algorithms: Introduction – Preliminaries – Lamport‘s algorithm – Ricart-Agrawala algorithm – Maekawa‘s algorithm – Suzuki–Kasami‘s broadcast algorithm. Deadlock detection in distributed systems: Introduction – System model – Preliminaries – Models of deadlocks – Knapp‘s classification – Algorithms for the single resource model, the AND model and the OR model.

3.1. DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS: INTRODUCTION

- Mutual exclusion is a fundamental problem in distributed computing systems.
- Mutual exclusion ensures that concurrent access of processes to a shared resource or data is serialized, that is, executed in mutually exclusive manner.

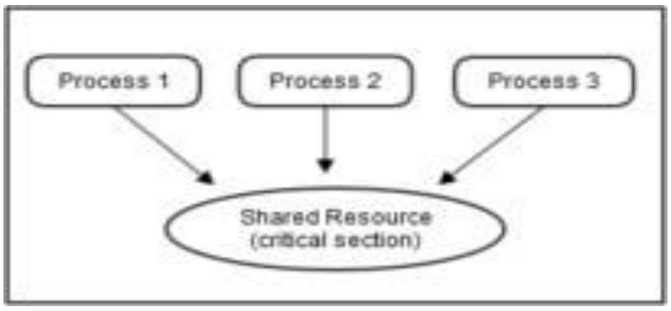


Figure 1: Three processes accessing a shared resource (critical section) simultaneously.

- Mutual exclusion in a distributed system states that only one process is allowed to execute the critical section (CS) at any given time.
- Message passing is the sole means for implementing distributed mutual exclusion.

There are three basic approaches for implementing distributed mutual exclusion:

1. Token based approach
2. Non-token based approach
3. Quorum based approach

1. In the **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 and it continues to hold the token until the execution of the CS is over. Mutual exclusion is ensured because the token is unique.

2. In the **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. A site enters the **Critical Section (CS)** when an assertion, defined on its local variables, becomes true. Mutual Exclusion is enforced because the assertion becomes true only at one site at any given time.

3. In the **quorum-based approach**, each site requests permission to execute the CS from a subset of sites (called a quorum). The quorums are formed in such a way that when two sites

concurrently request access to the CS, one site receives both the requests and which is responsible to make sure that only one request executes the CS at any time.

OBJECTIVES OF MUTUAL EXCLUSION ALGORITHMS

1. **Guarantee mutual exclusion** (required)
2. **Freedom from deadlocks** (desirable)
3. **Freedom from starvation** -- every requesting site should get to enter CS in a finite time (desirable)
4. **Fairness** -- requests should be executed in the order of arrivals, which would be based on logical clocks (desirable)
5. **Fault tolerance** -- failure in the distributed system will be recognized and therefore not cause any unduly prolonged disruptions (desirable)

3.1.1. PRELIMINARIES

We describe here,

1. **System model,**
2. **Requirements that mutual exclusion algorithms**
3. **Metrics we use to measure the performance of mutual exclusion algorithms.**

1. SYSTEM MODEL

- The system consists of N sites, **S1, S2, ..., SN.** We assume that a single process is running on each site.
- The process at site Si is denoted by pi.
- A process wishing to enter the CS, requests all other or a subset of processes by sending REQUEST messages, and waits for appropriate replies before entering the CS. While waiting the process is not allowed to make further requests to enter the CS.
- *A site can be in one of the following three states*:
    1. **Requesting the Critical Section.**
    2. **Executing the Critical Section.**
    3. **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 the token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS. Such state is referred to as 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.

Note:

- We assume that channels reliably deliver all messages, sites do not crash, and the network does not get partitioned.

- Some mutual exclusion algorithms are designed to handle such situations. Many algorithms use Lamport-style logical clocks to assign a timestamp to critical section requests.
- Timestamps are used to decide the priority of requests in case the of a conflict.
- A general rule followed is that the **smaller the timestamp of a request**, the **higher its priority to execute the CS.**
- We use the following notations:
  - **N** denotes the **number of processes or sites** involved in invoking the critical section,
  - **T** denotes the average **Message Time Delay**,
  - and **E** denotes the average critical section **Execution Time**.

2. REQUIREMENTS OF MUTUAL EXCLUSION ALGORITHMS

A mutual exclusion algorithm should satisfy the following properties:

a. 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.
b. 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. 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.
c. Fairness: Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the CS.

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

3. PERFORMANCE METRICS

The performance of mutual exclusion algorithms is generally measured by the following four metrics:
a. Message complexity: It is the number of messages that are required per CS execution by a site.
b. Synchronization delay: After a site leaves the CS, it is the time required and before the next site enters the CS (sees Figure 9.1).

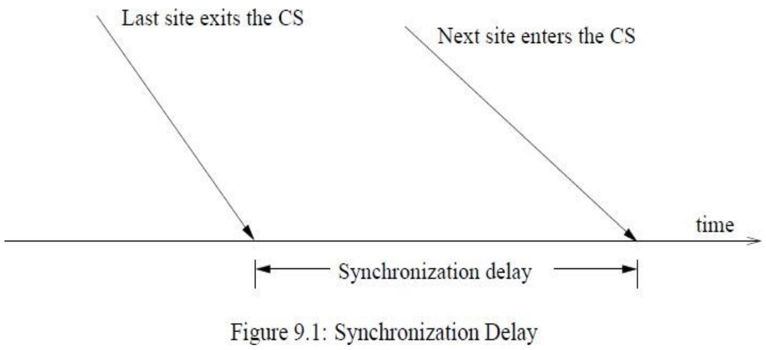


Figure 9.1: Synchronization Delay

c. Response time: It is the time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 9.2).

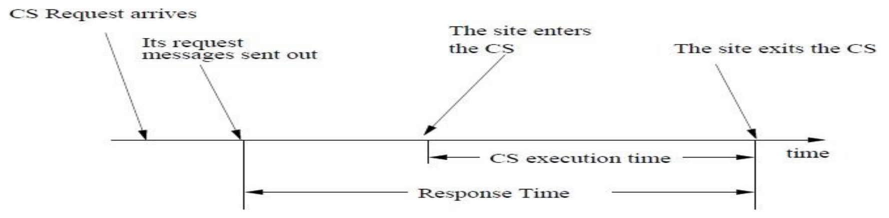


Figure 9.2: Response Time

Figure 2: Response Time

d. System throughput: It 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:

System Throughput=1/(SD+E)

Generally, the value of a performance metric fluctuates statistically from request to request and we generally consider the average value of such a metric.

Low and High Load Performance: The load is determined by the arrival rate of CS execution requests. Two special loading conditions, viz., **“low load"** and **“high load".**

- 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.

.

3.2. LAMPORT’S ALGORITHM

- The algorithm is fair in the sense that a request for CS is executed in the order of their timestamps and time is determined by logical clocks.
- When a site processes a request for the CS, it updates its local clock and assigns the request a timestamp.
- The algorithm executes CS requests in the increasing order of timestamps.
- Every site Si keeps a queue, **request_queuei**,

This algorithm requires communication channels to deliver messages the FIFO order.

The Algorithm

1. Requesting the critical section:

- When a site Si wants to enter the CS, it broadcasts a **REQUEST(tsi, i)** message to all other sites and places the request on **request_queuei.** ((tsi, i) denotes the timestamp of the request.)

- When a site Sj receives the **REQUEST(tsi, i)** message from site Si, places site Si’s Request on request_queuej and it returns a time stamped REPLY message to Si.

2. Executing the critical section:

Site Si enters the CS when the following two conditions hold:

L1: Si has received a message with timestamp larger than (tsi, i) from all other sites.

L2: Si's request is at the top of request_queuei.

3. Releasing the critical section:

- Site Si, upon exiting the CS, removes its request from the top of its request queue and broadcasts a time stamped RELEASE message to all other sites.
- When a site Sj receives a RELEASE message from site Si, it removes Si'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. Clearly, when a site receives a REQUEST, REPLY or RELEASE message, it updates its clock using the timestamp in the message.

Correctness

Theorem 1: ***Lamport's algorithm achieves mutual exclusion.***

Proof: **Proof is by contradiction**. Suppose two sites Si and Sj 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 Si and Sj have their own requests at the top of their request_queues and condition L1 hold at them. Without loss of generality, assume that Si's request has smaller timestamp than the request of Sj. 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_queuej when Sj was executing its CS. This implies that Sj's own request is at the top of its own request_queue when a smaller timestamp request, Si's request, is present in there quest_queuej – a contradiction!! Hence, Lamport's algorithm achieves mutual exclusion.

Theorem 2: ***Lamport's algorithm is fair.***

Proof: A distributed mutual exclusion algorithm is fair if the requests for CS are executed in the order of their timestamps. The proof is by contradiction. Suppose a site Si's request has a smaller timestamp than the request of another site Sj and Sj is able to execute the CS before Si. For Sj to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time say t, Sj 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_queueat a site is ordered by timestamp, and according to our assumption Si has lower timestamp. So Si's request must be placed ahead of the Sj's request in the request_queuej . This is a contradiction. Hence Lamport's algorithm is a fair mutual exclusion algorithm.

An Example

In Figures 9.3 to 9.6, we illustrate the operation of Lamport's algorithm. In Figure 9.3, sites S1 andS2 are making requests for the CS and send out REQUEST messages to other sites. The time stamps of the requests are (1, 1) and (1, 2), respectively. In Figure 9.4, both the sites S1 and S2 have received REPLY messages from all other sites. S1 has its request at the top of its request_queue but site S2 does not have its request at the top of its request_queue. Consequently, site S1 enters the CS. In Figure 9.5, S1 exits and sends RELEASE messages to all other sites. In Figure 9.6, site S2 has received REPLY from all other sites and also received a RELEASE message from siteS1. Site S2 updates its request_queue and its request is now at the top of its request_queue. Consequently, it enters the CS next.

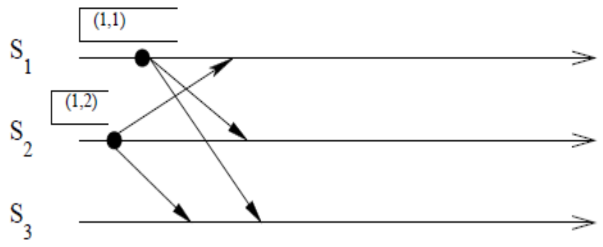


Figure 9.3: Sites S_1 and S_2 are Making Requests for the CS.

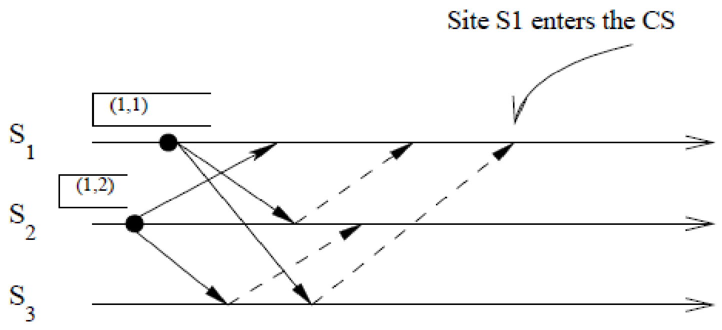


Figure 9.4: Site S_1 enters the CS.

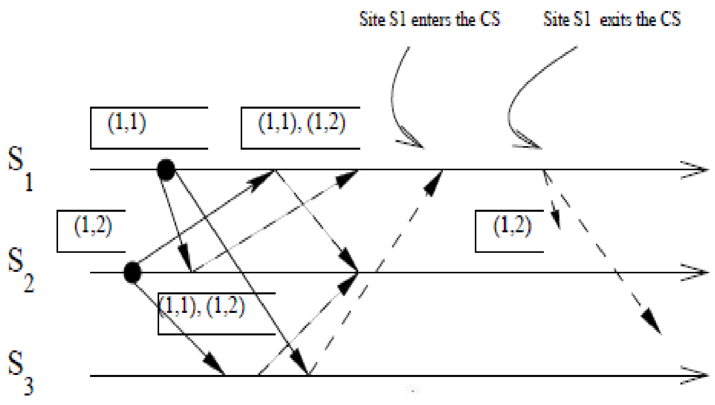


Figure 9.5: Site S_1 exits the CS and sends RELEASE messages.

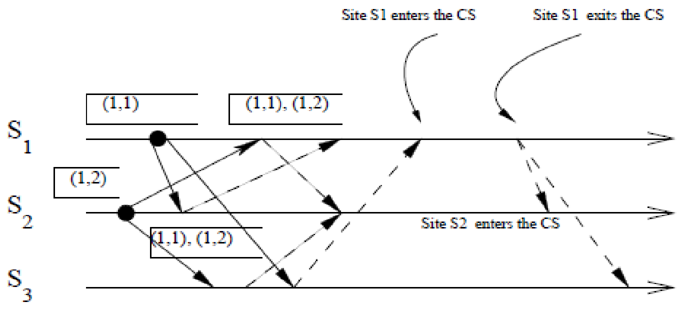


Figure 9.6: Site S_2 enters the CS

Performance

for each CS invocation

(N-1) REQUEST
(N-1) REPLY
(N-1) RELEASE,

Total 3(N-1) messages, synchronization delay Sd = average delay

3.3. 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. Timestamps are used to decide the priority of requests in case of conflict – if a process pi that is waiting to execute the critical section, receives a REQUEST message from process pj, then if the priority of pj's request is lower, pi defers the REPLY to pj and sends a REPLY message to pj only after executing the CS for it spending request.

- Otherwise, pi sends a REPLY message to pj immediately, provided it is currently not executing the CS. Thus, if several processes are requesting execution of the CS, the highest priority request succeeds in collecting all the needed REPLY messages and gets to execute the CS.

Each process pi maintains the **Request-Deferred array**, RDi, the size of which is the same as the number of processes in the system. Initially, ∀i ∀j: RDi[j]=0. Whenever pi defer the request sent by pj, it sets RDi[j]=1 and after it has sent a REPLY message to pj, it sets RDi[j]=0.

Note: Deferred – Postponed the request / waiting

ALGORITHM

1. Requesting the critical section:

(a) When a site Si wants to enter the CS, it broadcasts a time stamped REQUEST message to all other sites.

(b) When site Sj receives a REQUEST message from site Si, it sends a REPLY message to Site Si if site Sj is neither requesting nor executing the CS, or if the site Sj is requesting And Si’s request’s timestamp is smaller than site Sj’s own request’s timestamp. otherwise, the reply is deferred and Sj sets RDj[i]=1

2. Executing the critical section:

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

3. Releasing the critical section:

(d) When site Si exits the CS, it sends all the deferred REPLY messages: ∀j if RDi[j]=1, then send a REPLY message to Sj and set RDi[j]=0.

When a site receives a message, it updates its clock using the timestamp in the message. Also, when a site takes up a request for the CS for processing, it updates its local clock and assigns a timestamp to the request. In this algorithm, a site’s REPLY messages are blocked only by sites which are requesting the CS with higher priority (i.e., smaller timestamp).Thus, when a site sends out differed REPLY messages, site with the next highest priority request receives the last needed REPLY message and enters the CS. Execution of the CS requests in this algorithm is always in the order of their timestamps.

An Example

Figures 9.7 to 9.10 illustrate the operation of Ricart-Agrawala algorithm. In Figure 9.7, sites S1 and S2 are making requests for the CS and send out REQUEST messages to other sites. The timestamps of the requests are (2, 1) and (1, 2), respectively. In Figure 9.8, S2 has received REPLY messages from all other sites and consequently, it enters the CS. In Figure 9.9, S2 exits the CS and sends a REPLY message to site S1. In Figure 9.10, site S1 has received REPLY from all other sites and enters the CS next.

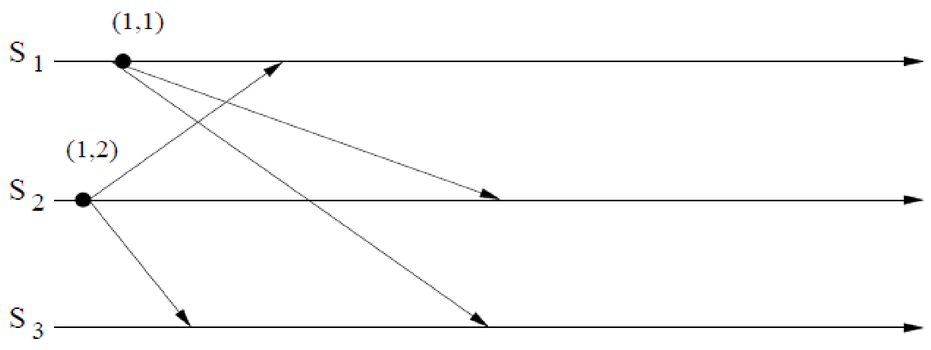


Figure 9.7: Sites S_1 and S_2 are making request for the CS

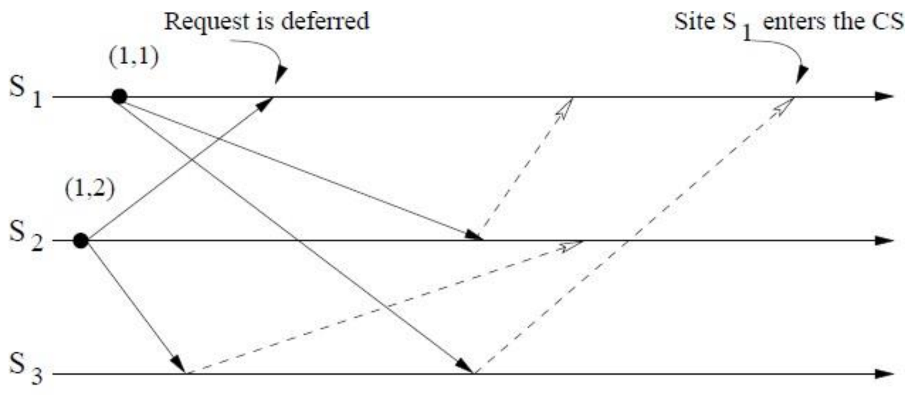


Figure 9.8: Site S_1 enters the CS

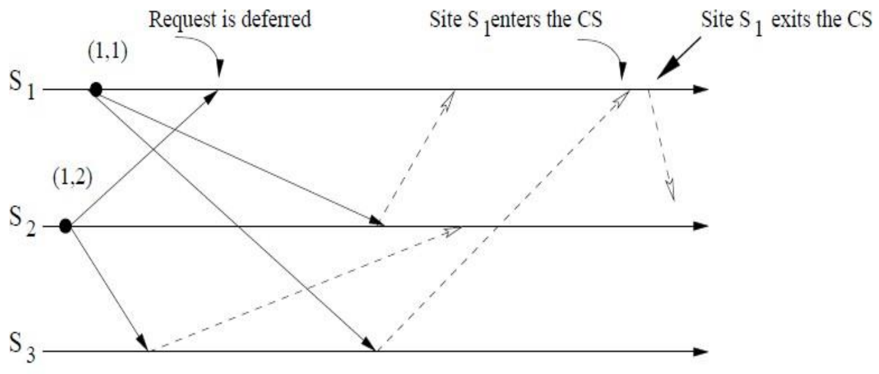


Figure 9.9: Site S_1 exits the CS and sends a REPLY message to S_2's deferred request

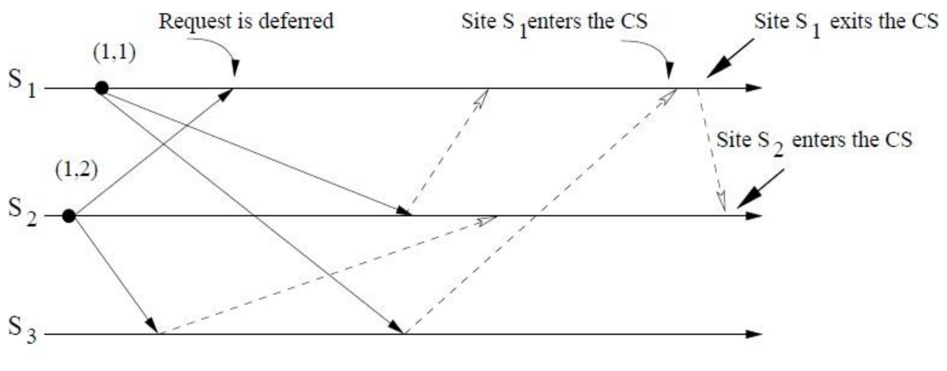


Figure 9.10: Site S_2 enters the CS

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.**

3.4. MAEKAWA'S ALGORITHM

Maekawa's Algorithm is quorum (subset) based approach to ensure mutual exclusion in distributed systems. As we know, In permission based algorithms like Lamport's Algorithm, Ricart-Agrawala Algorithm etc. a site request permission from every other site but in quorum based approach, A site does not request permission from every other site but from a subset of sites which is called **quorum**.

In this algorithm:

- Three type of messages (REQUEST, **REPLY** and **RELEASE**) are used.
- A site send a **REQUEST** message to all other site in its request set or quorum to get their permission to enter critical section.
- A site send a **REPLY** message to requesting site to give its permission to enter the critical section.
- A site send a **RELEASE** message to all other site in its request set or quorum upon exiting the critical section.

Maekawa's algorithm was the 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 |Ri| = √N. Since there is at least one common site between the request sets of any two sites (condition M1), every pair of sites has a common site which mediates conflicts between the pair. A site can have only one outstanding REPLY message at any time; that is, it grants permission to an incoming request if it has not granted permission to some other site. Therefore, mutual exclusion is guaranteed. This algorithm requires delivery of messages to be in the order they are sent between very pair of sites. Conditions M1 and M2 are necessary for correctness; whereas conditions M3 and M4 provide other desirable features to the algorithm. ConditionM3 states that the size of the requests sets of all sites must be equal implying that all sites should have to do equal amount of work to invoke mutual exclusion. Condition M4 enforces that exactly the same number of sites should request permission from any site implying that all sites have "equal responsibility" in granting permission to other sites.

ALGORITHM

In Maekawa's algorithm, a site Si executes the following steps to execute the CS.

1. Requesting the critical section

(a) A site Si requests access to the CS by sending REQUEST(i) messages to all sites in its request set Ri.

(b) When a site Sj receives the REQUEST (i) message, it sends a REPLY(j) message to Si 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.

2. Executing the critical section

(c) Site Si executes the CS only after it has received a REPLY message from every site in Ri.

3. Releasing the critical section

(d) After the execution of the CS is over, site Si sends a RELEASE (i) message to every site in Ri.

(e) When a site Sj receives a RELEASE(i) message from site Si, 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 3: *Maekawa's algorithm achieves mutual exclusion.*

Proof: Proof is by contradiction. Suppose two sites Si and Sj are concurrently executing the CS. This means site Si received a REPLY message from all sites in Ri and concurrently site Sj was able to receive a REPLY message from all sites in Rj. If Ri ∩ Rj= {Sk}, then site Sk must have sent REPLY messages to both Si and Sj concurrently, which is a contradiction. 2

Performance
Note that the size of a request set is √N. Therefore, an execution of the CS requires √N REQUEST,√N REPLY, and √N RELEASE messages, resulting in 3√N messages per CS execution. Synchronization delay in this algorithm is 2T. This is because after a site Si exits the CS, it first releases all the sites in Ri 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. As discussed next, Maekawa's algorithm is deadlock-prone. Measures to handle deadlocks require additional messages.

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 [14, 22]. Thus, a site may send a REPLY message to a site and later force a higher priority request from another site to wait.
Without the loss of generality, assume three sites Si, Sj, and Sk simultaneously invoke mutual exclusion. Suppose Ri ∩ Rj= {Sij}, Rj∩ Rk= {Sjk}, and Rk∩ Ri= {Ski}. Since sites do not send REQUEST messages to the sites in their request sets in any particular order and message delays are arbitrary, the following scenario is possible: Sij has been locked by Si (forcing Sj to wait at Sij), Sjk has been locked by Sj(forcing Sk to wait at Sjk), and Ski has been locked by Sk(forcing Si to wait at Ski). This state represents a deadlock involving sites Si, Sj, and Sk.

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 (unless the former has succeeded in acquiring locks on all the needed sites). A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request arrive sand waits at a site because the site has sent a REPLY message to a lower priority request. Deadlock handling requires the following three types of messages:

Deadlock handling requires the following three types of messages:

FAILED: A FAILED message from site Si to site Sj indicates that Si cannot grant Sj's request because it has currently granted permission to a site with a higher priority request.
INQUIRE: An INQUIRE message from Si to Sj indicates that Si would like to find out from Sjif it has succeeded in locking all the sites in its request set.
YIELD: A YIELD message from site Si to Sj indicates that Si is returning the permission to Sj(to yield to a higher priority request at Sj).

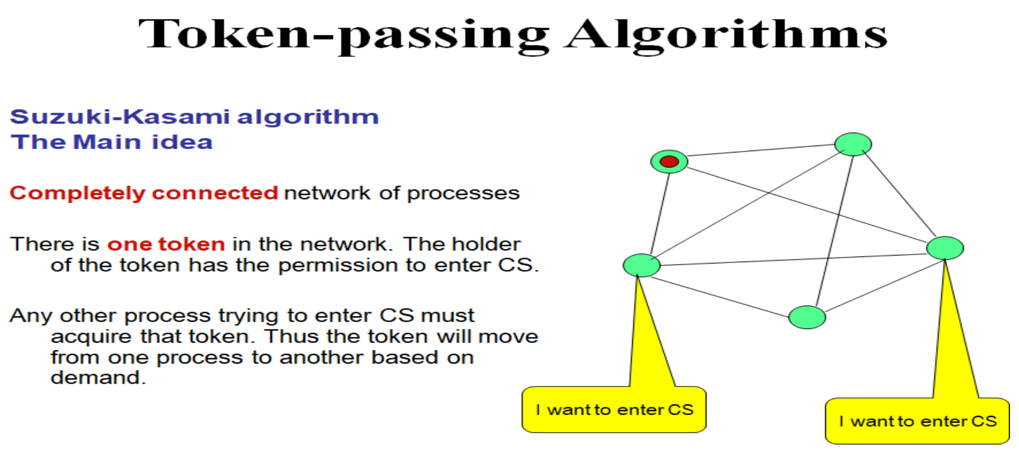
Details of how Maekawa's algorithm handles deadlocks are as follows:

• When a REQUEST(ts, i) from site Si blocks at site Sj because Sj has currently granted permission to site Sk, then Sj sends a FAILED(j) message to Si if Si's request has lower priority. Otherwise, Sj sends an INQUIRE(j) message to site Sk.• In response to an INQUIRE(j) message from site Sj, site Sk sends a YIELD(k) message to Sj provided Sk has received a FAILED message from a site in its request set or if it sent a YIELD to any of these sites, but has not received a new GRANT from it.• In response to a YIELD(k) message from site Sk, site Sj assumes as if it has been released by Sk, places the request of Sk at appropriate location in the request queue, and sends a GRANT(j) to the top request's site in the queue.
Thus, Maekawa-type algorithms require extra messages to handle deadlocks and may exchange these messages even though there is no deadlock. Maximum number of messages required per CS execution in this case is 5√N.

3.5 SUZUKI-KASAMI'S BROADCAST ALGORITHM

- **Suzuki–Kasami algorithm** is a token-based algorithm for achieving mutual exclusion in distributed systems.
- In token-based algorithms, A site is allowed to enter its critical section if it possesses the unique token.
- Non-token-based algorithms uses timestamp to order requests for the critical section where as sequence number is used in token based algorithms.

- Each request for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.



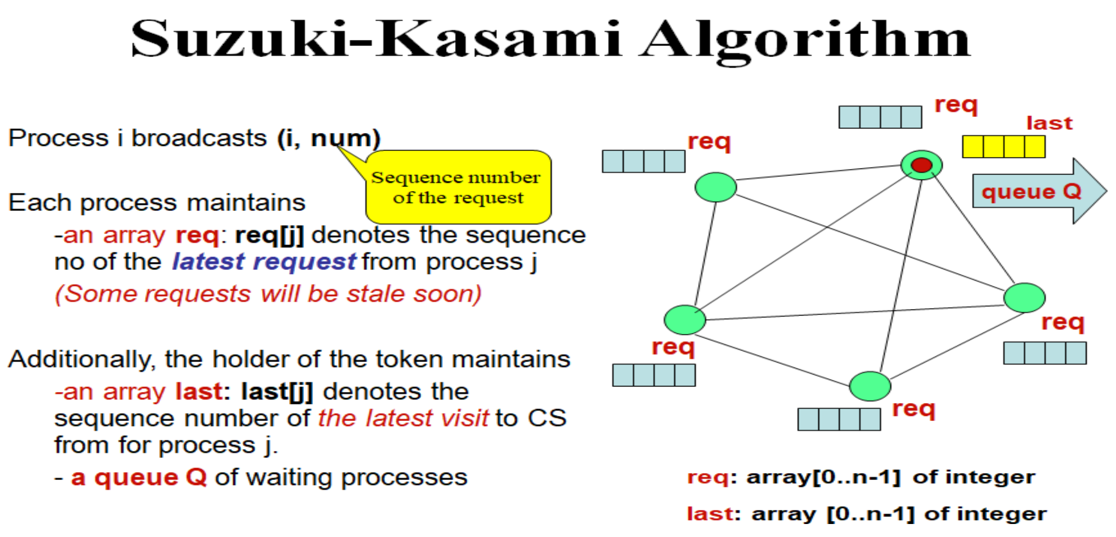
Token-passing Algorithms

Suzuki-Kasami algorithm
The Main idea

Completely connected network of processes

There is one token in the network. The holder of the token has the permission to enter CS.

Any other process trying to enter CS must acquire that token. Thus the token will move from one process to another based on demand.



Suzuki-Kasami Algorithm

Process i broadcasts **(i, num)**

Each process maintains

-an array **req**: **req[j]** denotes the sequence no of the ***latest request*** from process j

(Some requests will be stale soon)

Additionally, the holder of the token maintains

-an array **last: last[j]** denotes the sequence number of *the latest visit* to CS from for process j.

- **a queue Q** of waiting processes

req: array[0..n-1] of integer

last: array [0..n-1] of integer

In Suzuki-Kasami's algorithm if a site that wants to enter the CS, does not have the token, it broadcasts a REQUEST message for the token to all other sites. A site which possesses the token sends it to the requesting site upon the receipt of its REQUEST message. If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of the CS.

The basic idea underlying this algorithm may sound rather simple, however, there are the following two design issues must be efficiently addressed:

1. How to distinguishing an outdated REQUEST message from a current REQUEST message:

Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied. If a site can not determined if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it. This will not violate the correctness; however, this may seriously degrade the performance by wasting messages and increasing the delay at sites that are genuinely requesting the token. Therefore, appropriate mechanisms should implemented to determine if a token request message is outdated.

2. **How to determine which site has an outstanding request for the CS:** After a site has finished the execution of the CS, it must determine what sites have an outstanding request for the CS so that the token can be dispatched to one of them. The problem is complicated because when a site Si receives a token request message from a site Sj, site Sj may have an outstanding request for the CS. However, after the corresponding request for the CS has been satisfied at Sj, an issue is how to inform site Si (and all other sites) efficiently about it. Outdated REQUEST messages are distinguished from current REQUEST messages in the following manner: A REQUEST message of site Sjhas the form REQUEST(j, n) where n (n=1, 2,
...) is a sequence number which indicates that site Sjis requesting its nth CS execution.
A site Si keeps an array of integers RNi[1..N] where RNi[j] denotes the largest sequence number received in a REQUEST message so far from site Sj. When site Si receives a REQUEST (j, n)message, it sets RNi[j]:=max(RNi[j], n). Thus, when a site Si receives a REQUEST (j, n) message, the request is outdated if RNi[j]>n. Sites with outstanding requests for the CS are

determined in the following manner: The token consists of a queue of requesting sites, Q, and an array of integers LN[1..N], where LN[j] is the sequence number of the request which site Sj executed most recently. After executing its CS, a site Si updates LN[i]:=RNi[i] to indicate that its request corresponding to sequence number RNi[i] has been executed. Token array LN[1..N] permits a site to determine if a site has an outstanding request for the CS. Note that at site Si if RNi[j]=LN[j]+1, then site Sj is currently requesting token. After executing the CS, a site checks this condition for all the j' s to determine all the sites which are requesting the token and places their id's in queue Q if these id's are not already present in the Q. Finally, the site sends the token to the site whose id is at the head of the Q.

ALGORITHM

1. Requesting the critical section

(a) If requesting site Si does not have the token, then it increments its sequence number, RNi[i], and sends a REQUEST(i, sn) message to all other sites. ('sn' is the updated value of RNi[i].)

(b) When a site Sj receives this message, it sets RNj[i] to max(RNj[i], sn). If Sj has the idle token, then it sends the token to Si if RNj[i]=LN[i]+1.

2. Executing the critical section

(c) Site Si executes the CS after it has received the token.

3. Releasing the critical section Having finished the execution of the CS, site Si takes the following actions:

(d) It sets LN[i] element of the token array equal to RNi[i].

(e) For every site Sj whose id is not in the token queue, it appends its id to the token queue if RNi[j]=LN[j]+1.

(f) If the token queue is nonempty after the above update, Si deletes the top site id from the token queue and sends the token to the site indicated by the id. Thus, after executing the CS, a site gives priority to other sites with outstanding requests for the CS (over its pending requests for the CS). Note that Suzuki-Kasami's algorithm is not symmetric because a site retains the token even if it does not have a request for the CS, which is contrary to the spirit of Ricart and Agrawala's definition of symmetric algorithm: *"no site possesses the right to access its CS when it has not been requested"*.

Correctness

Mutual exclusion is guaranteed because there is only one token in the system and a site holds the token during the CS execution.

Theorem: *A requesting site enters the CS in finite time.*

Proof: Token request messages of a site Si reach other sites in finite time. Since one of these sites will have token in finite time, site Si's request will be placed in the token queue in finite time. Since there can be at most N − 1 requests in front of this request in the token queue, site Si will get the token and execute the CS in finite time. 2

Performance
Beauty of Suzuki-Kasami algorithm lies in its simplicity and efficiency. No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request. If a site does not hold the token when it makes a request, the algorithm requires N messages to obtain the token. Synchronization delay in this algorithm is 0 or T.

3.6. DEADLOCK DETECTION IN DISTRIBUTED SYSTEMS

INTRODUCTION

A deadlock is a condition where a process cannot proceed because it needs to obtain a resource held by another process and it itself is holding a resource that the other process needs.

We can consider two types of deadlock:

1. Communication deadlock occurs when process A is trying to send a message to process B, which is trying to send a message to process C which is trying to send a message to A.
2. A resource deadlock occurs when processes are trying to get exclusive access to devices, files, locks, servers, or other resources. We will not differentiate between these types of deadlock since we can consider communication channels to be resources without loss of generality.

"A deadlock can be defined as a condition where a set of processes request resources that are held by other processes in the set."

Deadlock deals with various components **like deadlock prevention, deadlock avoidance other then deadlock detection**.

- **Deadlock prevention** is commonly achieved by either having a process acquire all the needed resources simultaneously before it begins execution or by pre-empting a process that hold the needed resource.
- In the **deadlock avoidance** approach to distributed system, a resource is granted to a process if the resulting global system is safe.
- **Deadlock detection** requires an examination of the status of the process-resources interaction for the presence of a deadlock condition. To resolve the deadlock, we have to abort a deadlocked process.

3.7. SYSTEM MODEL

- A distributed system consists of a set of processors that are connected by a communication network.
- The communication delay is finite but unpredictable.
- A distributed program is composed of a set of n asynchronous processes p1, p2, . . . , pi, . . . , pn that communicates by message passing over the communication network.
- Without loss of generality we assume that each process is running on a different processor.
- The processors do not share a common global memory and communicate solely by passing messages over the communication network.

- The communication medium may deliver messages out of order, messages may be lost garbled or duplicated due to timeout and retransmission, processors may fail and communication links may go down.
- The system can be modelled as a directed graph in which vertices represent the processes and edge represent unidirectional communication channels.

We make the following assumptions:

- The systems have only reusable resources.
- Processes are allowed to make only exclusive access to resources.
- There is only one copy of each resource.

A process can be in two states: ***running*** **or** ***blocked***. In the running state (also called *active* state), a process has all the needed resources and is either executing or is ready for execution. In the blocked state, a process is waiting to acquire some resource.

Wait-For-Graph (WFG)

- In distributed systems, the state of the system can be modelled by **directed graph, called a *wait for graph* (WFG)**.
- In a WFG, nodes are processes and there is a directed edge from node P1 to mode P2 if P1 is blocked and is waiting for P2 to release some resource.
- A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.

Figure 10.1 shows a WFG, where process P11 of site 1 has an edge to process P21 of site 1 and P32 of site 2 is waiting for a resource which is currently held by process P21. At the same time process P32 is waiting on process P33 to release a resource. If P21 is waiting on process P11, then processes P11, P32 and P21 form a cycle and all the four processes are involved in a deadlock depending upon the request model.

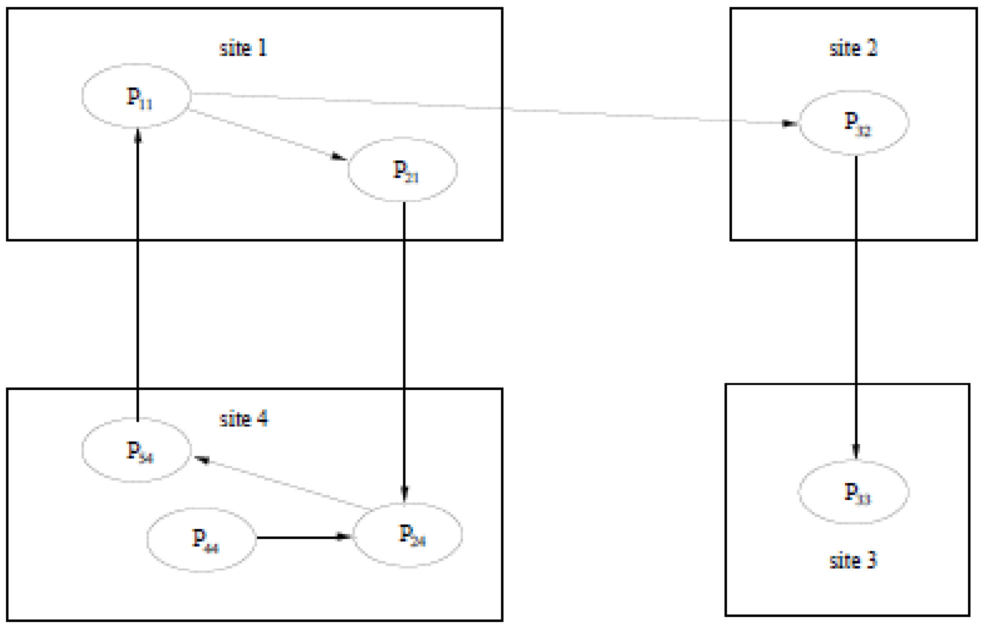


Figure 10.1: Example of a WFG

3.8. PRELIMINARIES

Deadlock Handling Strategies

There are three strategies for handling deadlocks,

1. **Deadlock Prevention,**
2. **Deadlock Avoidance,**
3. **Deadlock Detection.**

Handling of deadlock becomes highly complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every inter site communication involves a finite and unpredictable delay. Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by pre-empting a process which holds the needed resource. This approach is highly inefficient and impractical in distributed systems. In deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system). However, due to several problems, deadlock avoidance is impractical in distributed systems.

Issues in Deadlock Detection

Deadlock handling using the approach of deadlock detection entails addressing two basic issues:

1. **Detection of existing deadlocks**
2. **Resolution of detected deadlocks.**

1. Detection of Deadlocks

- Detection of deadlocks involves addressing two issues: maintenance of the WFG and searching of the WFG for the presence of cycles (or knots). Since in distributed systems, a cycle or knot may involve several sites, the search for cycles greatly depends upon how the WFG of the system is represented across the system. Depending upon the way WFG information is maintained and search for cycles is carried out, there are centralized, distributed, and hierarchical algorithms for deadlock detection in distributed systems .

Correctness Criteria: A deadlock detection algorithm must satisfy the following two conditions:

(i) Progress (No undetected deadlocks): The algorithm must detect all existing deadlocks in finite time. Once a deadlock has occurred, the deadlock detection activity should continuously progress until the deadlock is detected. In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.

(ii) Safety (No false deadlocks): The algorithm should not report deadlocks which do not exist (called *phantom or false* deadlocks). In distributed systems where there is no global memory and there is no global clock, it is difficult to design a correct deadlock detection algorithm because sites may obtain out of date and inconsistent WFG of the system. As a result, sites may detect a cycle which never existed but whose different segments existed in the system at different times. This is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

2. Resolution of a Detected Deadlock

Deadlock resolution involves breaking existing wait-for dependencies between the processes to resolve the deadlock. It involves rolling back one or more deadlocked processes and assigning their resources to blocked processes so that they can resume execution. Note that several deadlock detection algorithms propagate information regarding wait-for dependencies along the edges of the wait-for graph. Therefore, when a wait-for dependency is broken, the corresponding information should be immediately cleaned from the system. If this information is not cleaned in timely manner, it may result in detection of phantom deadlocks. Untimely and inappropriate cleaning of broken wait-for dependencies is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

3.9. MODELS OF DEADLOCKS

- Distributed systems allow many kinds of resource requests. A process might require a single resource or a combination of resources for its execution. This section introduces a hierarchy of request models starting with very restricted forms to the ones with no restrictions whatsoever. This hierarchy shall be used to classify deadlock detection algorithms based on the complexity of the resource requests they permit.

The Single Resource Model

The single resource model is the simplest resource model in a distributed system, here a process can have at most one outstanding request for only one unit of a resource. Since the maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock. In a later section, an algorithm to detect deadlock in the single resource model is presented.

The AND Model

In the AND model, a process can request for more than one resource simultaneously and the
request is satisfied only after all the requested resources are granted to the process. The requested resources may exist at different locations. The out degree of a node in the WFG for AND model can be more than 1. The presence of a cycle in the WFG indicates a deadlock in the AND model. Each node of the WFG in such a model is called an AND node. Consider the example WFG described in the Figure 10.1. Process P11 has two outstanding resource requests. In case of the AND model, P11shall become active from idle state only after both the resources are granted. There is a cycle P11->P21->P24->P54->P11 which corresponds to a deadlock situation.

In the AND model, if a cycle is detected in the WFG, it implies a deadlock but not vice versa. That is, a process may not be a part of a cycle, it can still be deadlocked. Consider process P44 in Figure 10.1. It is not a part of any cycle but is still deadlocked as it is dependent on P24 which is deadlocked. Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

The OR Model

In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted. The requested resources may exist at different locations. If all requests in the WFG are OR requests, then the nodes are called OR nodes. Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model. To make it more clear, consider Figure 10.1. If all nodes are OR nodes, then process P11 is not deadlocked because once process P33 releases its resources, P32 shall become active as one of its requests is satisfied. After P32 finishes execution and releases its resources, process P11 can continue with its processing.

In the OR model, the presence of a knot indicates a deadlock. In a WFG, a vertex v is in a knot if for all u:: u is reachable from v : v is reachable from u. No paths originating from a knot shall have dead ends.

A deadlock in the OR model can be intuitively defined as follows : A process Pi is blocked if it has a pending OR request to be satisfied. With every blocked process, there is an associated set of processes called dependent set. A process shall move from *idle* to *active* state on receiving a grant message from any of the processes in its dependent set. A process is permanently blocked if it never receives a grant message from any of the processes in its dependent set. Intuitively, a set of processes S is deadlocked if all the processes in S are permanently blocked. To formally state that a set of processes is deadlocked, the following conditions hold true:

1. Each of the process is the set S is blocked,
2. The dependent set for each process in S is a subset of S, and
3. No grant message is in transit between any two processes in set S.

We now show that a set of processes S shall remain permanently blocked in the OR model if

the above conditions are met. A blocked process P is the set S becomes *active* only after receiving a grant message from a process in its dependent set, which is a subset of S. Note that no grant message can be expected from any process in S because they are all blocked. Also, the third condition states that no grant messages in transit between any two processes in set S. So, all the processes in set S are permanently blocked.
Hence, deadlock detection in the OR model is equivalent to finding knots in the graph. Note that, there can be a process deadlocked which is not a part of a knot. Consider the Figure 10.1 where P44 can be deadlocked even though it is not in a knot. So, in an OR model, a blocked process P is deadlocked if it is either in a knot or it can only reach processes on a knot.

The AND-OR Model

A generalization of the previous two models (OR model and AND model) is the AND-OR model. In the AND-OR model, a request may specify any combination of *and* and *or* in the resource request. For example, in the AND-OR model, a request for multiple resources can beof the form x *and* (y *or* z). The requested resources may exist at different locations. To detectthe presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG. Since a deadlock is a stable property (i.e., once it exists, it does not go away by itself), this property can be exploited and a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock. However, this is a very inefficient strategy. Efficient algorithms to detect deadlocks in AND-OR model are discussed in Herman
.

The $\binom{p}{q}$ Model

Another form of the AND-OR model is the $\binom{p}{q}$ model (called the P-out-of-Q model) which allows a request to obtain any k available resources from a pool of n resources. Both the models are the same in expressive power. However, $\binom{p}{q}$ model lends itself to a much more compact formation of a request.

Every request in the $\binom{p}{q}$ model can be expressed in the AND-OR model and vice-versa. Note that AND requests for p resources can be stated as $\binom{p}{p}$ and OR requests for p resources can be stated as $\binom{p}{1}$.

Unrestricted Model

In the unrestricted model, no assumptions are made regarding the underlying structure of resource requests. In this model, only one assumption that the deadlock is stable is made and hence it is the most general model. This way of looking at the deadlock problem helps in separation of concerns: Concerns about properties of the problem (stability and deadlock) are separated from underlying distributed systems computations (e.g., message passing versus synchronous communication). Hence, these algorithms can be used to detect other stable properties as they deal with this general model. But, these algorithms are of more theoretical value for distributed systems since no further assumptions are made about the underlying distributed systems computations which leads to a great deal of overhead.

3.10. KNAPP'S CLASSIFICATION OF DISTRIBUTED DEADLOCK DETECTION

Algorithms

- Distributed deadlock detection algorithms can be divided into four classes : path-pushing, edge-chasing, diffusion computation, and global state detection.

Path-Pushing Algorithms

In path-pushing algorithms, distributed deadlocks are detected by maintaining an explicit global WFG. The basic idea is to build a global WFG for each site of the distributed system. In this class of algorithms, at each site whenever deadlock computation is performed, it sends its local WFG to all the neighboring sites. After the local data structure of each site is updated, this updated WFG is then passed along to other sites, and the procedure is repeated until some site has a sufficiently complete picture of the global state to announce deadlock or to establish that no deadlocks are present.

Edge-Chasing Algorithms

In an edge-chasing algorithm, the presence of a cycle in a distributed graph structure is be verified by propagating special messages called probes, along the edges of the graph. These probe messages are different than the request and reply messages. The formation of cycle can be deleted by a site if it receives the matching probe sent by it previously.
Whenever a process that is executing receives a probe message, it simply discards this message and continues. Only blocked processes propagate probe messages along their outgoing edges. An interesting variation of this method can be found in Mitchell [36], where probes are sent upon request and in the opposite direction of the edges.
Main advantage of edge-chasing algorithms is that probes are fixed size messages which is normally very short. Examples of such algorithms include Chandy et al., Choudhary et al., Kshemkalyani-Singhal , and Sinha-Natarajan algorithms.

Diffusing Computations Based Algorithms

In *diffusion computation* based distributed deadlock detection algorithms; deadlock detection computation is diffused through the WFG of the system. These algorithms make use of echo algorithms to detect deadlocks. This computation is superimposed on the underlying distributed computation. If this computation terminates, the initiator declares a deadlock. The main feature of the superimposed computation is that the global WFG is implicitly reflected in the structure of the computation. The actual WFG is never built explicitly.
To detect a deadlock, a process sends out query messages along all the outgoing edges in the WFG. These queries are successively propagated (i.e., diffused) through the edges of the WFG. Queries are discarded by a running process and are echoed back by blocked processes in the following way: When a blocked process receives first query message for a particular deadlock detection initiation, it does not send a reply message until it has received a reply message for every query it sent (to its successors in the WFG). For all subsequent queries for this deadlock detection initiation, it immediately sends back a reply message. The initiator of a deadlock detection detects a deadlock when it receives reply for every query it had sent out. Examples of these types of deadlock detection algorithms are Chandy-Misra-Haas algorithm for OR model and Chandy-Herman algorithm .

Global State Detection Based Algorithms

Global state detection based deadlock detection algorithms exploit the following facts: (i) A consistent snapshot of a distributed system can be obtained without freezing the underlying computation and (ii) a consistent snapshot may not represent the system state at any moment in time, but if a stable property holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot.
Therefore, distributed deadlocks can be detected by taking a snapshot of the system and examining it for the condition of a deadlock. Examples of these types of algorithms include Bracha- Toueg , Wang et al., and Kshemkalyani-Singhal algorithms.

3.11. MITCHELL AND MERRITT'S ALGORITHM FOR THE SINGLE-RESOURCE MODEL

Mitchell and Merritt's algorithm belongs to the class of edge-chasing algorithms where probes are sent in opposite direction of the edges of WFG. When a probe initiated by a process comes back to it, the process declares deadlock. The algorithm has many good features like:
1. Only one process in a cycle detects the deadlock. This simplifies the deadlock resolution – this process can abort itself to resolve the deadlock. This algorithm can be improvised by including priorities and the lowest priority process in a cycle detects deadlock and aborts.
2. In this algorithm process which is detected in deadlock is aborted spontaneously, even though under this assumption phantom deadlocks cannot be excluded. It can be shown, however, that only genuine deadlocks will be detected in the absence of spontaneous aborts.

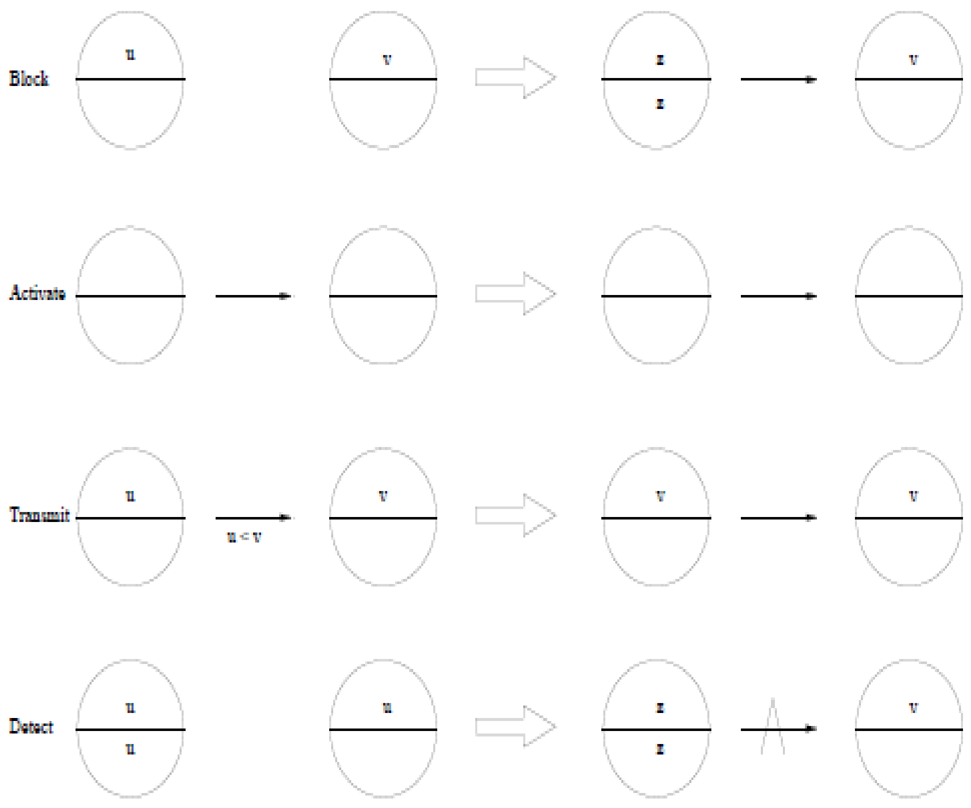


Figure 10.2: The four possible state transitions

Each node of the WFG has two local variables, called labels: a private label, which is unique

to the node at all times, though it is not constant, and a public label, which can be read by other processes and which may not be unique. Each process is represented as u/v where u and u are the public and private labels, respectively. Initially, private and public labels are equal for each process.
A global WFG is maintained and it defines the entire state of the system.The Algorithm is defined by the four state transitions shown in Figure 10.2, where z = inc(u, v), and inc(u, v) yields a unique label greater than both u and v labels that are not shown do not change. Block creates an edge in the WFG. Two messages are needed, one resource request and one message back to the blocked process to inform it of the public label of the process it is waiting for. Activate denotes that a process has acquired the resource from the process it was waiting for. Transmit propagates larger labels in the opposite direction of the edges by sending a probe message. Whenever a process receives a probe which is less then its public label, then it simply ignores that probe. Detect means that the probe with the private label of some process has returned to it, indicating a deadlock.
Mitchell andMerritt showed that every deadlock is detected. Next, we show that in the absence of spontaneous aborts, only genuine deadlocks are detected. As there are no spontaneous aborts, we have following invariant:

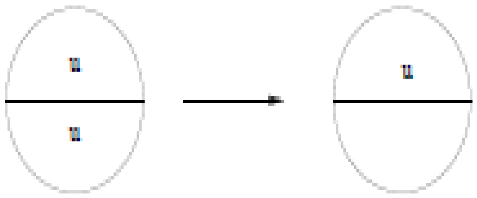
For all processes u/v: u <= v
Proof. Initially u = u for all processes. The only requests that change u or v are
1) Block: u and v are set such that u = v.
(2) Transmit: u is increased.
Hence, the invariant.

From the previous invariant, we have the following lemmas.

Lemma 12. *For any process u/v, if u > u, then u was set by a Transmit step.*

Theorem 13. *If a deadlock is detected, a cycle of blocked nodes exists.*

Proof. A deadlock is detected if the following edge p →p' exists:



We will prove the following claims:
(1) u has been propagated from p to p' via a sequence of Transmits.
(2) P has been continuously blocked since it transmitted u.
(3) All intermediate nodes in the transmit path of (l), including p', have been continuously blocked since they transmitted u.

From the above claims, the proof for the theorem follows as discussed below:

From the invariant and the uniqueness of private label u of p' : u < v. By Lemma 4.1, u was set by a Transmit step. From the semantics of Transmit, there is some p" with private label u and public label w. If w = u, then p" = p, and it is a success. Otherwise, if w <u, we repeat the argument. Since there are only processes, one of them is p. If p is active then it indicates that it has transmitted u else it is blocked if it detects deadlock. Hence upon blocking it incremented its private label. But then private and public labels cannot be equal. Consider a process which has been active since it transmitted u. Clearly, its predecessor is also active, as Transmits migrate in opposite direction. By repeating this argument, we can show p has been active since it transmitted u. The above algorithm can be easily extended to include priorities where whenever a deadlock occurs, the lowest priority process gets aborted. This algorithm has two phases. The first phase is almost identical to the algorithm. In the second phase the smallest priority is propagated around the circle, The propagation stops when one process recognizes the propagated priority as its own.

Message Complexity
Now we calculate the complexity of the algorithm. If we assume that a deadlock persists long enough to be detected, the worst-case complexity of the algorithm is s(s - 1)/2 Transmit steps,
where s is the number of processes in the cycle.

3.12. CHANDY-MISRA-HAAS ALGORITHM FOR THE AND MODEL

We now discuss Chandy-Misra-Haas's distributed deadlock detection algorithm for AND model that is based on edge-chasing. The algorithm uses a special message called *probe*, which is a triplet (i, j, k), denoting that it belongs to a deadlock detection initiated for process Pi and it is being sent by the home site of
process Pj to the home site of process Pk. A probe message travels along the edges of the global WFG graph, and a deadlock is detected when a probe message returns to the process that initiated it.
A process Pj is said to be *dependent* on another process Pk if there exists a sequence of processes Pj,Pi1, Pi2, ..., Pim, Pk such that each process except Pk in the sequence is blocked and each process, except the Pj, holds a resource for which the previous process in the sequence is waiting. Process Pj is said to be *locally dependent* upon process Pk if Pj is dependent upon Pk and both the processes are on the same site.

Data Structures
Each process Pi maintains a boolean array, dependent i, where dependent i(j) is true only if Pi knows that Pj is dependent on it. Initially, dependent i(j) is false for all i and j.

The Algorithm

The following algorithm is executed to determine if a blocked process is deadlocked:

```
if $P_i$ is locally dependent on itself
    then declare a deadlock
    else for all $P_j$ and $P_k$ such that
      (a) $P_i$ is locally dependent upon $P_j$, and
      (b) $P_j$ is waiting on $P_k$, and
      (c) $P_j$ and $P_k$ are on different sites,
    send a probe (i, j, k) to the home site of $P_k$
```

```
On the receipt of a probe (i, j, k), the site takes
    the following actions:

if
    (d) $P_k$ is blocked, and
    (e) $dependent_k$(i) is false, and
    (f) $P_k$ has not replied to all requests $P_j$,
    then
      begin
```

```
$dependent_k$(i) = true;
if k=i
    then declare that $P_i$ is deadlocked
else for all $P_m$ and $P_n$ such that
    (a') $P_k$ is locally dependent upon $P_m$, and
    (b') $P_m$ is waiting on $P_n$, and
    (c') $P_m$ and $P_n$ are on different sites,
    send a probe (i, m, n) to the home site of $P_n$
end.
```

Therefore, a probe message is continuously circulated along the edges of the global WFG graph and a deadlock is detected when a probe message returns to its initiating process.

Performance Analysis

In the algorithm, one probe message (per deadlock detection initiation) is sent on every edge of the WFG which that two sites. Thus, the algorithm exchanges at most m(n − 1)/2 messages to detect a deadlock that involves m processes and that spans over n sites. The size of messages is fixed and is very small (only 3 integer words). Delay in detecting a deadlock is O(n).

3.13. CHANDY-MISRA-HAAS ALGORITHM FOR THE OR MODEL

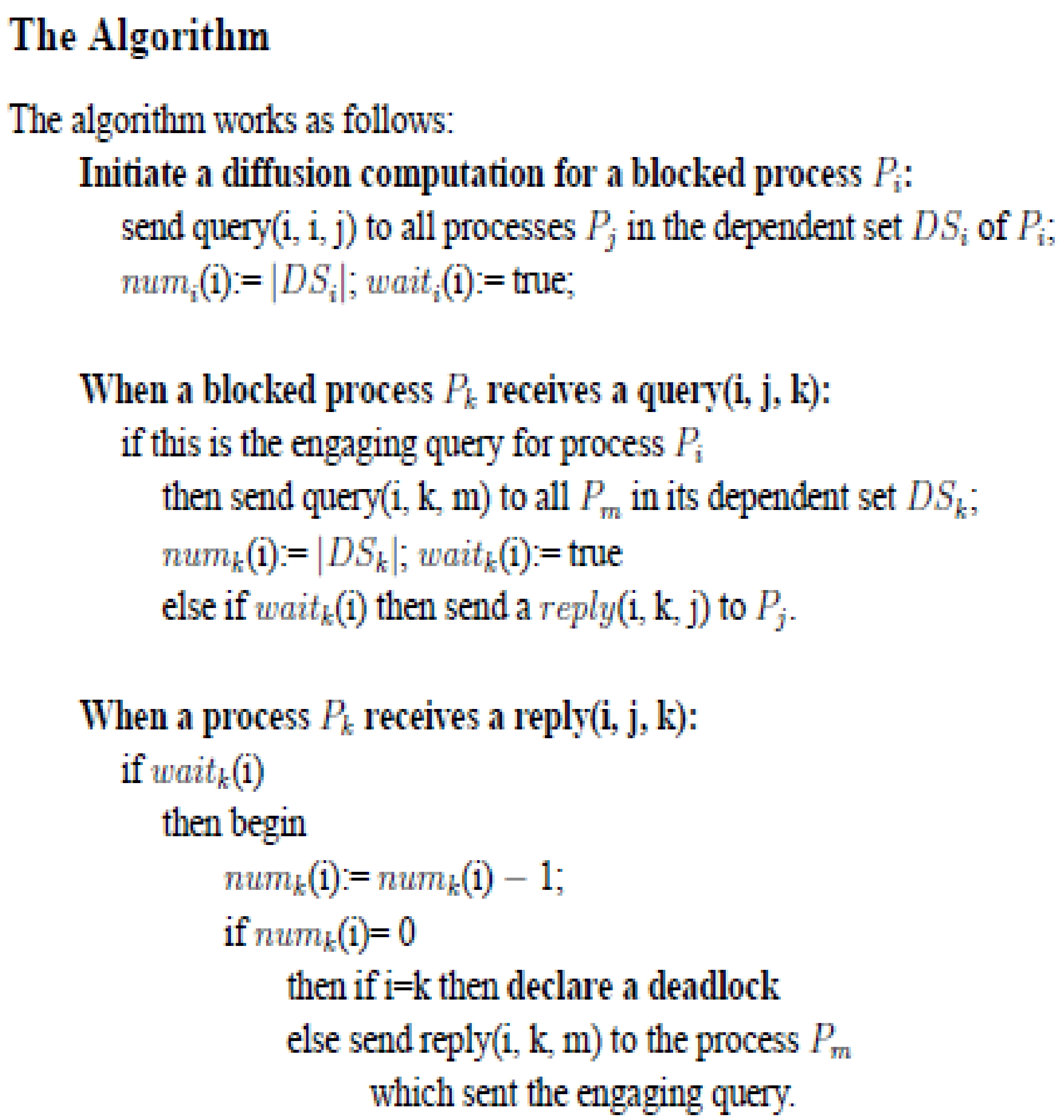
We now discuss Chandy-Misra-Haas distributed deadlock detection algorithm for OR model that is based on the approach of diffusion-computation. A blocked process determines if it is deadlocked by initiating a diffusion computation. Two types of messages are used in a diffusion computation: query(i, j, k) and reply(i, j, k), denoting that they belong to a diffusion computation initiated by a process Pi and are being sent from process Pj to process Pk.

Basic Idea

A blocked process initiates deadlock detection by sending query messages to all processes in its dependent set (i.e., processes from which it is waiting to receive a message). If an active process receives a query or reply message, it discards it. When a blocked process Pk receives a query(i, j,k) message, it takes the following actions:

1. If this is the first query message received by Pk for the deadlock detection initiated by Pi (called the *engaging query*), then it propagates the query to all the processes in its dependent set and sets a local variable numk(i) to the number of query messages sent.

2. If this is not the engaging query, then Pk returns a reply message to it immediately provided Pk has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query. Process Pk maintains a boolean variable waitk(i) that denotes the fact that it has been continuously blocked since it received the last engaging query from process Pi. When a blocked process Pk receives a reply(i, j, k) message, it decrements numk(i) only if waitk(i) holds. A process sends a reply message in response to an engaging query only after it has received a reply to every query message it had sent out for this engaging query. The initiator process detects a deadlock when it receives reply messages to all the query messages it had sent out.



The Algorithm

The algorithm works as follows:

```
**Initiate a diffusion computation for a blocked process $P_i$:**
    send query(i, i, j) to all processes $P_j$ in the dependent set $DS_i$ of $P_i$;
    $num_i(i) := |DS_i|$; $wait_i(i) :=$ true;

**When a blocked process $P_k$ receives a query(i, j, k):**
    if this is the engaging query for process $P_i$
        then send query(i, k, m) to all $P_m$ in its dependent set $DS_k$;
        $num_k(i) := |DS_k|$; $wait_k(i) :=$ true
        else if $wait_k(i)$ then send a $reply$(i, k, j) to $P_j$.

**When a process $P_k$ receives a reply(i, j, k):**
    if $wait_k(i)$
        then begin
            $num_k(i) := num_k(i) - 1$;
            if $num_k(i) = 0$
                then if i=k then declare a deadlock
                else send reply(i, k, m) to the process $P_m$
                    which sent the engaging query.
```

For ease of presentation, we assumed that only one diffusion computation is initiated for a process. In practice, several diffusion computations may be initiated for a process (A diffusion computation is initiated every time the process gets blocked), but, at any time only one diffusion computation is current for any process. However, messages for outdated diffusion computations may still be in transit. The current diffusion computation can be distinguished from outdated ones by using sequence numbers.

Performance Analysis

For every deadlock detection, the algorithm exchanges e query messages and e reply messages, where e=n(n-1) is the number of edges.