CHAPTER

9 Distributed mutual exclusion algorithms

9.1 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 a mutually exclusive manner. Mutual exclusion in a distributed system states that only one process is allowed to execute the critical section (CS) at any given time. In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion. Message passing is the sole means for implementing distributed mutual exclusion. The decision as to which process is allowed access to the CS next is arrived at by message passing, in which each process learns about the state of all other processes in some consistent way. The design of distributed mutual exclusion algorithms is complex because these algorithms have to deal with unpredictable message delays and incomplete knowledge of the system state. There are three basic approaches for implementing distributed mutual exclusion:

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

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. The algorithms based on this approach essentially differ in the way a site carries out the search for the token. 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. 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, at least one site receives both the requests and this site is responsible to make sure that only one request executes the CS at any time.

In this chapter, we describe several distributed mutual exclusion algorithms and compare their features and performance. We discuss relationship among various mutual exclusion algorithms and examine trade-offs among them.

9.2 Preliminaries

In this section, we describe the underlying system model, discuss the requirements that mutual exclusion algorithms should satisfy, and discuss what metrics we use to measure the performance of mutual exclusion algorithms.

9.2.1 System model

The system consists of N sites, S_1, $S_2, \ldots, S_N$. Without loss of generality, we assume that a single process is running on each site. The process at site S_i is denoted by p_i. All these processes communicate asynchronously over an underlying communication network. 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: requesting the CS, executing the CS, or neither requesting nor executing the CS (i.e., idle). In the "requesting the CS" state, the site is blocked and cannot 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 refereed 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.

We do not make any assumption regarding communication channels if they are FIFO or not. This is algorithm specific. 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 of a conflict. The general rule followed is that the smaller the timestamp of a request, the higher its priority to execute the CS.

We use the following notation: N denotes the number of processes or sites involved in invoking the critical section, T denotes the average message delay, and E denotes the average critical section execution time.

9.2.2 Requirements of mutual exclusion algorithms

A mutual exclusion algorithm should satisfy the following properties:

1. **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.
2. **Liveness property** This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages that will never arrive. In addition, a site must not wait indefinitely to execute the CS while other sites are repeatedly executing the CS. That is, every requesting site should get an opportunity to execute the CS in finite time.
3. **Fairness** Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the CS. In mutual exclusion algorithms, the fairness property generally means that the CS execution requests are executed in order of their arrival in the system (the time is determined by a logical clock).

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

9.2.3 Performance metrics

The performance of mutual exclusion algorithms is generally measured by the following four metrics:

- **Message complexity** This is the number of messages that are required per CS execution by a site.
- **Synchronization delay** After a site leaves the CS, it is the time required and before the next site enters the CS (see Figure 9.1). Note that normally one or more sequential message exchanges may be required after a site exits the CS and before the next site can enter the CS.
- **Response time** This is the time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 9.2). Thus, response time does not include the time a request waits at a site before its request messages have been sent out.
- **System throughput** This is the rate at which the system executes requests for the CS. If SD is the synchronization delay and E is the average critical section execution time, then the throughput is given by the following equation:

$$\text{System throughput} = \frac{1}{(SD + E)}.$$

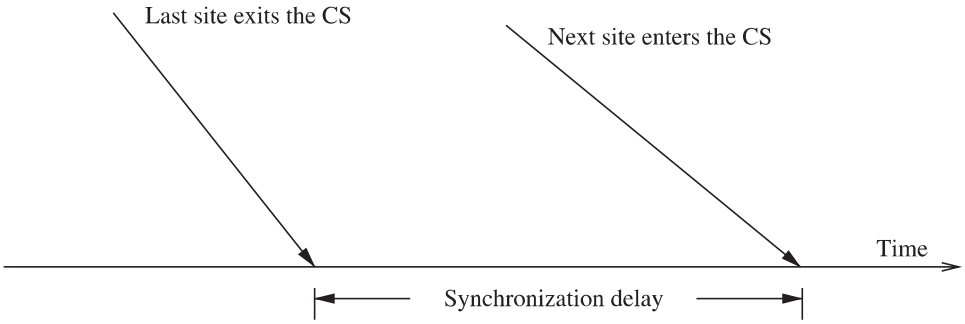

Figure 9.1 Synchronization delay.

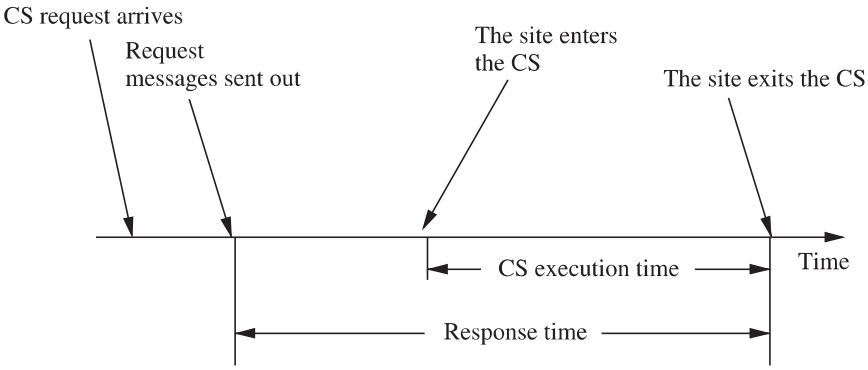

Figure 9.2 Response time.

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. Performance of a mutual exclusion algorithm depends upon the load and we often study the performance of mutual exclusion algorithms under 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. Thus, in heavy load conditions, after having executed a request, a site immediately initiates activities to execute its next CS request. A site is seldom in the idle state in heavy load conditions. For many mutual exclusion algorithms, the performance metrics can be computed easily under low and heavy loads through a simple mathematical reasoning.

Best and worst case performance

Generally, mutual exclusion algorithms have best and worst cases for the performance metrics. In the best case, prevailing conditions are such that a performance metric attains the best possible value. For example, in most

mutual exclusion algorithms the best value of the response time is a round-trip message delay plus the CS execution time, $2T + E$. Often for mutual exclusion algorithms, the best and worst cases coincide with low and high loads, respectively. For examples, the best and worst values of the response time are achieved when load is, respectively, low and high; in some mutual exclusion algorithms the best and the worse message traffic is generated at low and heavy load conditions, respectively.

9.3 Lamport's algorithm

Lamport developed a distributed mutual exclusion algorithm (Algorithm 9.1) as an illustration of his clock synchronization scheme [12]. The algorithm is fair in the sense that a request for CS are 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 S_i keeps a queue, $request\_queue_i$, which contains mutual exclusion requests ordered by their timestamps. (Note that this queue is different from the queue that contains local requests for CS execution awaiting their turn.) This algorithm requires communication channels to deliver messages in FIFO order.

Requesting the critical section

- When a site S_i wants to enter the CS, it broadcasts a REQUEST(ts_i, i) message to all other sites and places the request on $request\_queue_i$. ((ts_i, i) denotes the timestamp of the request.)
- When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it places site S_i's request on $request\_queue_j$ and returns a timestamped REPLY message to S_i.

Executing the critical section

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

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

L2: S_i's request is at the top of $request\_queue_i$.

Releasing the critical section

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

Algorithm 9.1 Lamport's algorithm.

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 9.1 *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! Hence, Lamport's algorithm achieves mutual exclusion. □

Theorem 9.2 *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 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 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. Hence Lamport's algorithm is a fair mutual exclusion algorithm. □

Example In Figures 9.3 to 9.6, we illustrate the operation of Lamport's algorithm. In Figure 9.3, sites S_1 and S_2 are making requests for the CS and send out REQUEST messages to other sites. The timestamps of the requests are (1,1) and (1,2), respectively. In Figure 9.4, both the sites S_1 and S_2 have received REPLY messages from all other sites. S_1 has its request at the top of its *request_queue* but site S_2 does not have its request at the top of its *request_queue*. Consequently, site S_1 enters the CS. In Figure 9.5, S_1 exits and sends RELEASE mesages to all other sites. In Figure 9.6, site S_2 has received REPLY from all other sites and also received a RELEASE message

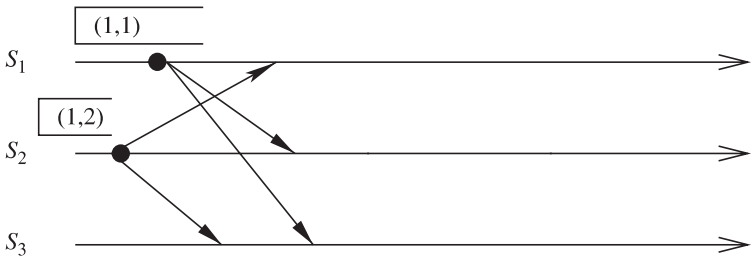

Figure 9.3 Sites S_1 and S_2 are Making Requests for the CS.

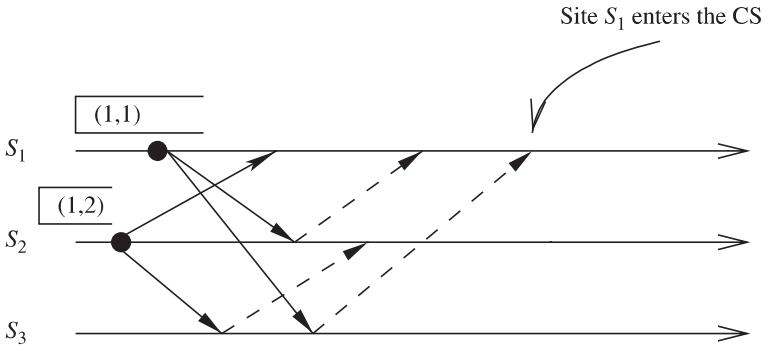

Figure 9.4 Site S_1 enters the CS.

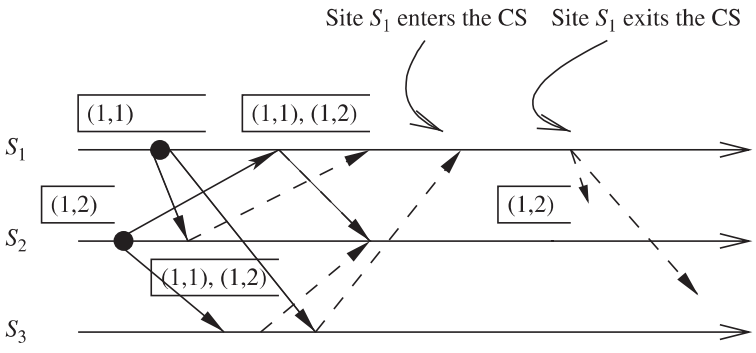

Figure 9.5 Site S_1 exits the CS and sends RELEASE messages.

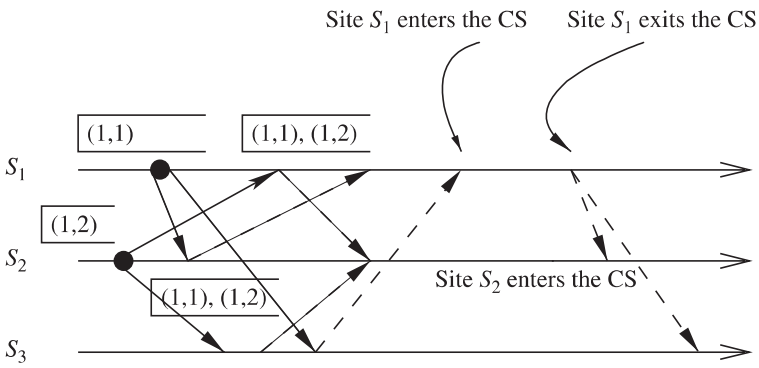

Figure 9.6 Site S_2 enters the CS.

from site S_1. Site S_2 updates its *request_queue* and its request is now at the top of its *request_queue*. Consequently, it enters the CS next.

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. The 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 a 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 a timestamp higher than its own, it can conclude that site S_j does not have any smaller timestamp request which is still pending (because communication channels preserves FIFO ordering).

With this optimization, Lamport's algorithm requires between $3(N-1)$ and $2(N-1)$ messages per CS execution.

9.4 Ricart–Agrawala algorithm

The Ricart–Agrawala [21] algorithm (Algorithm 9.2) assumes that 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 p_i that is waiting to execute the critical section receives a REQUEST message from process p_j, then if the priority of p_j's request is lower, p_i defers the REPLY to p_j and sends a REPLY message to p_j only after executing the CS for its pending request. Otherwise, p_i sends a REPLY message to p_j 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 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 defers 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 $RD_i[j] = 0$.

Requesting the critical section

(a) When a site S_i wants to enter the CS, it broadcasts a timestamped REQUEST message to all other sites.

(b) When site S_j receives a REQUEST message from site S_i, it sends a REPLY message to site S_i if site S_j is neither requesting nor executing the CS, or if the site S_j is requesting and S_i's request's timestamp is smaller than site S_j's own request's timestamp. Otherwise, the reply is deferred and S_j sets $RD_j[i] := 1$.

Executing the critical section

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

Releasing the critical section

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

Algorithm 9.2 The Ricart–Agrawala algorithm.

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 that are requesting the CS with higher priority (i.e., smaller timestamp). Thus, when a site sends out deferred REPLY messages, the 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.

Correctness

Theorem 9.3 *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 (i.e., smaller timestamp) than the request of S_j. Clearly, S_i received S_j's request after it has made its own request. (Otherwise, S_i's request will have lower priority.) 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, the Ricart–Agrawala algorithm achieves mutual exclusion. □

In the Ricart–Agrawala algorithm, for every requesting pair of sites, the site with higher priority request will always defer the request of the lower priority site. At any time only the highest priority request succeeds in getting all the needed REPLY messages.

Example Figures 9.7 to 9.10 illustrate the operation of the Ricart–Agrawala algorithm. In Figure 9.7, sites S_1 and S_2 are each making requests for the

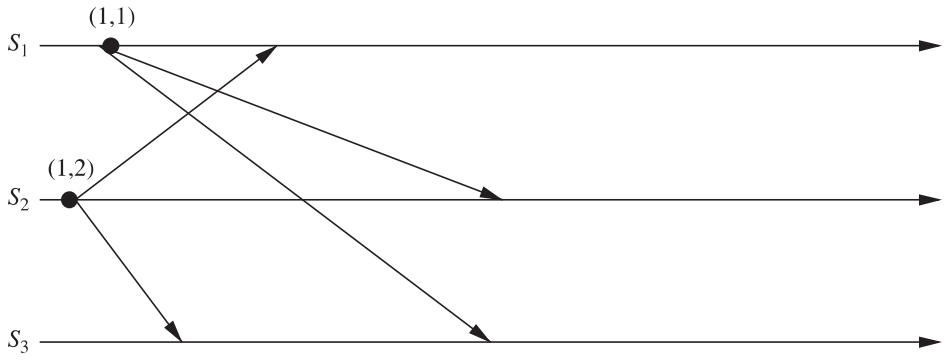

Figure 9.7 Sites S_1 and S_2 each make a request for the CS.

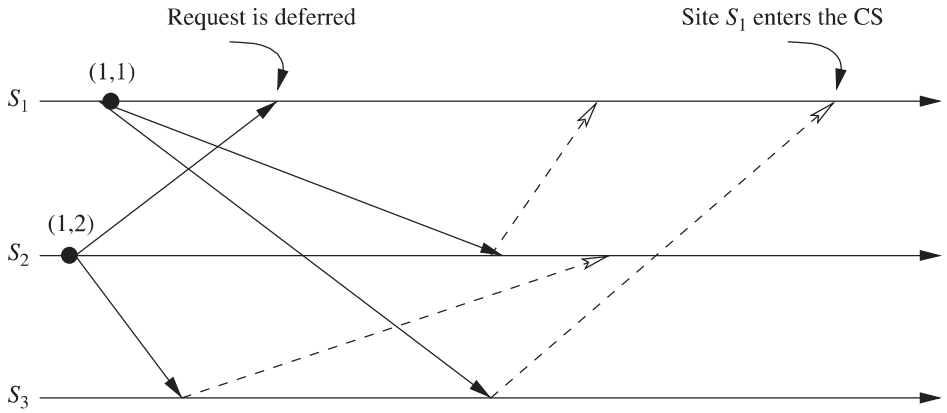

Figure 9.8 Site S_1 enters the CS.

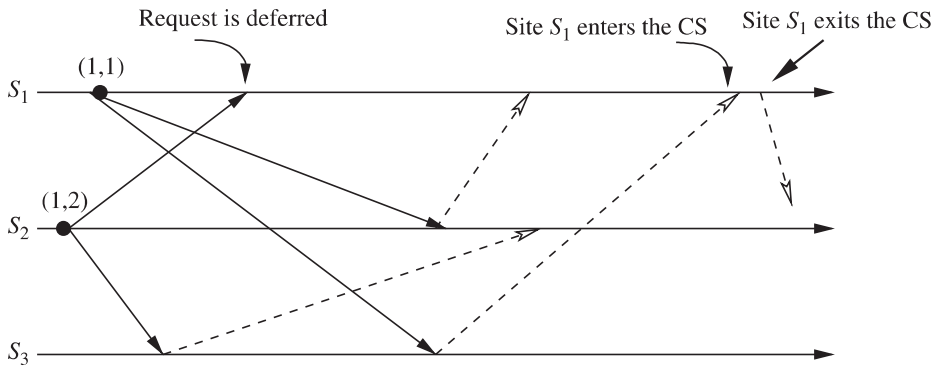

Figure 9.9 Site S_1 exits the CS and sends a REPLY message to S_2's deferred request.

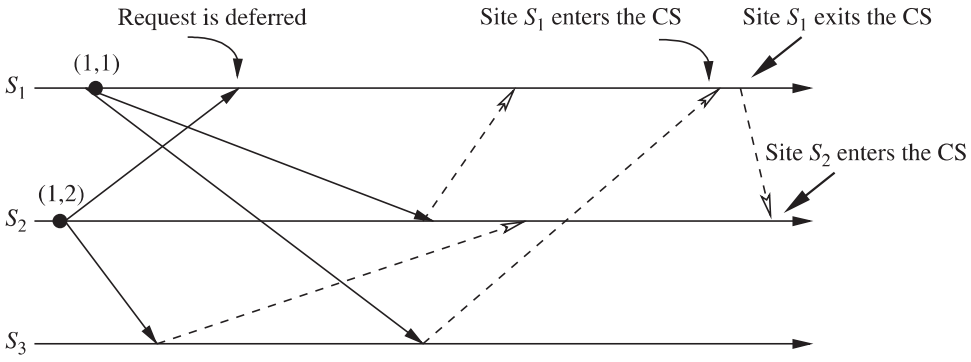

Figure 9.10 Site S_2 enters the CS.

CS and sending out REQUEST messages to other sites. The timestamps of the requests are (2,1) and (1,2), respectively. In Figure 9.8, S_2 has received REPLY messages from all other sites and, consequently, enters the CS. In Figure 9.9, S_2 exits the CS and sends a REPLY mesage to site S_1. In Figure 9.10, site S_1 has received REPLY from all other sites and enters the CS next.

Performance

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

9.5 Singhal's dynamic information-structure algorithm

Most mutual exclusion algorithms use a static approach to invoke mutual exclusion, i.e., they always take the same course of actions to invoke mutual exclusion no matter what is the state of the system. A problem with these algorithms is the lack of efficiency because these algorithms fail to exploit the changing conditions in the system. Note that an algorithm can exploit dynamic conditions of the system to optimize the performance.

For example, if few sites are invoking mutual exclusion very frequently and other sites invoke mutual exclusion much less frequently, then a frequently invoking site need not ask for the permission of less frequently invoking site every time it requests an access to the CS. It only needs to take permission from all other frequently invoking sites. Singhal [28] developed an adaptive mutual exclusion algorithm based on this observation. The information-structure of the algorithm evolves with time as sites learn about the state of the system through messages. Dynamic information-structure mutual exclusion

algorithms are attractive because they can adapt to fluctuating system conditions to optimize the performance.

The design of such adaptive mutual exclusion algorithms is challenging and we list some of the design challenges next:

- How does a site efficiently know what sites are currently actively invoking mutual exclusion?
- When a less frequently invoking site needs to invoke mutual exclusion, how does it do it?
- How does a less frequently invoking site makes a transition to more frequently invoking site and vice-versa?
- How do we ensure that mutual exclusion is guaranteed when a site does not take the permission of every other site?
- How do we ensure that a dynamic mutual exclusion algorithm does not waste resources and time in collecting systems state, offsetting any gain?

System model

We consider a distributed system consisting of n autonomous sites, say S_1, $S_2, \ldots, S_n$, which are connected by a communication network. We assume that the sites communicate completely by message passing. Message propagation delay is finite but unpredictable and, between any pair of sites, messages are delivered in the order they are sent. For the ease of presentation, we assume that the underlying communication network is reliable and sites do not crash. However, methods have been proposed for recovery from message losses and site failures.

Data structures

The information-structure at a site S_i consists of two sets. The first set R_i, called the *request set*, contains the sites from which S_i must acquire permission before executing CS. The second set I_i, called the *inform set*, contains the sites to which S_i must send its permission to execute CS after executing its CS.

Every site S_i maintains a logical clock C_i, which is updated according to Lamport's rules. Every request for CS execution is assigned a timestamp which is used to determine its priority. The smaller the timestamp of a request, the higher its priority. Every site maintains three boolean variables to denote the state of the site: *Requesting*, *Executing*, and *My_priority*. *Requesting* and *Executing* are true if and only if the site is requesting or executing CS, respectively. *My_priority* is true if the pending request of S_i has priority over the current incoming request.

Initialization
The system starts in the following initial state:

For a site S_i ($i = 1$ to n),
$R_i := \{S_1, S_2, \ldots, S_i - 1, S_i\}$
$I_i := S_i$
$C_i := 0$
$Requesting = Executing :=$ False

Thus, initially site S_i, $1 \leq i \leq n$, sends request messages only to sites S_i, $S_i - 1, \ldots, S_1$. If we stagger sites S_n to S_1 from left to right, then the initial system state has the following two properties:

1. Each site requests permission from all the sites to its right and from no site to its left. Conversely, for a site, all the sites to its left asks for its permission and no site to its right asks for its permission. Or putting together, for a site, only all the sites to its left will ask for its permission and it will ask for the permission of only all the sites to its right. Therefore, every site S_i divides all the sites into two disjoint groups: all the sites in the first group request permission from S_i, and S_i requests permission from all the sites in the second group. This property is important for enforcing mutual exclusion.
2. The cardinality of R_i decreases in a stepwise manner from left to right. Due to this reason, this configuration has been called "staircase pattern" in topological sense [26].

9.5.1 Description of the algorithm

Site S_i executes the three steps shown in Algorithm 9.3 to invoke mutual exclusion. The REQUEST message handler at a site processes incoming REQUEST messages. It takes actions such as updating the information-structure and sending REQUEST/REPLY messages to other sites. The REQUEST message handler at site S_i is given in Algorithm 9.3. The REPLY message handler at a site processes incoming REPLY messages. It updates the information-structure. The REPLY message handler at site S_i is given in Algorithm 9.3. Note that the REQUEST and REPLY message handlers and the steps of the algorithm access shared data structures, viz., C_i, R_i, and I_i. To guarantee the correctness, it's important that execution of the REQUEST and REPLY message handlers and all three steps of the algorithm (except "wait for $R_i = \emptyset$ to hold" in step 1) mutually exclude each other.

Step 1: (Request critical section)

Requesting = true;
C_i:=$C_i + 1$;
Send REQUEST(C_i, i) message to all sites in R_i;
Wait until $R_i = \emptyset$; /* Wait until all sites in R_i have sent a reply to S_i */
Requesting := false;

Step 2: (Execute critical section)

Executing := true;
Execute CS;
Executing := false;

Step 3: (Release critical section)

For every site S_k in I_i (except S_i) do
Begin
$I_i := I_i - \{S_k\}$;
Send REPLY(C_i, i) message to S_k;
$R_i := R_i + \{S_k\}$
End

REQUEST message handler:

/* Site S_i is handling message REQUEST(c, j) */
$C_i := max\{C_i, c\}$;
Case

Requesting = true:
Begin
if *My_priority* then $I_i := I_i + \{S_j\}$
/**My_Priority* is true if the pending request of S_i has priority over the incoming request */
Else
Begin
Send REPLY(C_i, i) message to S_j;
If not ($S_j \in R_i$) then
Begin
$R_i := R_i + \{S_j\}$;
Send REQUEST(C_i, i) message to site S_j;
End;
End;
End;

Executing = true: $I_i := I_i + \{S_j\}$;
Executing = false $\wedge$ Requesting = false:
Begin
$R_i := R_i + \{S_j\}$;
Send REPLY(C_i, i) message to S_j;
End;

REPLY message handler:

/* Site S_i is handling a message REPLY(c, j) */
Begin
C_i:= max{C_i, c};
$R_i := R_i - \{S_j\}$;
End;

Algorithm 9.3 Singhal's dynamic information-structure algorithm [28].

An explanation of the algorithm

At high level, S_i acquires permission to execute the CS from all sites in its request set R_i and it releases the CS by sending a REPLY message to all sites in its inform set I_i.

If site S_i, which itself is requesting the CS, receives a higher priority REQUEST message from a site S_j, then S_i takes the following actions: (i) S_i immediately sends a REPLY message to S_j, (ii) if S_j is not in R_i, then[1] S_i also sends a REQUEST message to S_j, and (iii) S_i places an entry for S_j in R_i. Otherwise (i.e., if the request of S_i has priority over the request of S_j), S_i places an entry for S_j into I_i so that S_j can be sent a REPLY message when S_i finishes with the execution of the CS.

If S_i receives a REQUEST message from S_j when it is executing the CS, then it simply puts S_j in I_i so that S_j can be sent a REPLY message when S_i finishes with the execution of the CS. If S_i receives a REQUEST message from S_j when it is neither requesting nor executing the CS, then it places an entry for S_j in R_i and sends S_j a REPLY message.

Rules for information exchange and updating request and inform sets are such that the staircase pattern is preserved in the system even after the sites have executed the CS any number of times. However, the positions of sites in the staircase pattern change as the system evolves. (For a proof of this, see [28].) The site to execute CS last positions itself at the right end of the staircase pattern.

9.5.2 Correctness

We informally discuss why the algorithm achieves mutual exclusion and why it is free from deadlocks. For a formal proof, the readers are referred to [27].

Achieving mutual exclusion

Note that the initial state of the information-structure satisfies the following condition: for every S_i and S_j, either $S_j \in R_i$ or $S_i \in R_j$. Therefore, if two sites request CS, one of them will always ask for the permission of the another. However, this is not sufficient for mutual exclusion [28]. Whenever there is a conflict between two sites (i.e., they concurrently invoke mutual exclusion), the sites dynamically adjust their request sets such that both request permission of each other satisfying the condition for mutual exclusion. This is a nice feature of the algorithm because if the information-structures of the sites satisfy the condition for mutual exclusion all the

[1] Absence of S_j from R_i implies that S_i has not previously sent a REQUEST message to S_j. This is the reason why S_i also sends a REQUEST message to S_j when it receives a REQUEST message from S_j. This step is also required to preserve the staircase pattern of the information-structure of the system.

time, the sites will exchange more messages. Instead, it is more desirable to dynamically adjust the request set of the sites as and when needed to insure mutual exclusion because it optimizes the number of messages exchanged.

Freedom from deadlocks

In the algorithm, each request is assigned a globally unique timestamp which determines its priority. The algorithm is free from deadlocks because sites use timestamp ordering (which is unique system wide) to decide request priority and a request is blocked only by higher priority requests.

Example Consider a system with five sites $S_1, \ldots, S_5$. Suppose S_2 and S_3 want to enter the CS concurrently, and they both send appropriate request messages. S_3 sends a request message to sites in its Request set – $\{S_1, S_2\}$, and S_2 sends a request message to the only site in its Request set – $\{S_1\}$. There are three possible scenarios:

1. If timestamp of S_3's request is smaller, then on receiving S_3's request, S_2 sends a REPLY message to S_3. S_2 also adds S_3 to its Request set and sends S_3 a REQUEST message. On receiving a REPLY message from S_2, S_3 removes S_2 from its Request set. S_1 sends a REPLY to both S_2 and S_3 because it is neither requesting to enter the CS nor executing the CS. S_1 adds S_2 and S_3 to its Request set because any one of these sites could possibly be in the CS when S_1 requests for an entry into CS in the future. On receiving S_1's REPLY message, S_3 removes S_1 from its Request set and, since it has REPLY messages from all sites in its (initial) Request set, it enters the CS.
2. If timestamp of S_3 is larger, then on receiving S_3's request, S_2 adds S_3 to its Inform set. When S_2 gets a REPLY from S_1, it enters the CS. When S_2 relinquishes the CS, it informs S_3 (the i.d. of S_3 is present in S_2's Inform set) about its consent to enter the CS. Then, S_2 removes S_3 from its Inform set and add S_3 to its Request set. This is logical because S_3 could be executing in CS when S_2 requests a "CS entry" permission in the future.
3. If S_2 receives a REPLY from S_1 and starts executing CS before S_3's REQUEST reaches S_2, S_2 simply adds S_3 to its Inform set, and sends S_3 a REPLY after exiting the CS.

9.5.3 Performance analysis

The synchronization delay in the algorithm is T. Below, we compute the message complexity in low and heavy loads.

Low load condition

In the case of low traffic of CS requests, most of the time only one or no request for the CS will be present in the system. Consequently, the staircase

pattern will re-establish between two sucssive requests for CS and there will seldom be an interference among the CS requests from different sites. In the staircase configuration, the cardinality of the request sets of the sites will be 1, 2, . . . , $(n-1)$, n, respectively, from right to left. Therefore, when the traffic of requests for CS is low, sites will send 0, 1, 2, . . . , $(n-1)$ number of REQUEST messages with equal likelihood (assuming uniform traffic of CS requests at sites). Therefore, the mean number of REQUEST messages sent per CS execution for this case will be $= (0+1+2+\ldots+(n-1))/n = (n-1)/2$. Since a REPLY message is returned for every REQUEST message, the average number of messages exchanged per CS execution will be $2^*(n-1)/2 = (n-1)$.

Heavy load condition

When the rate of CS requests is high, all the sites will always have a pending request for CS execution. In this case, a site receives on average $(n-1)/2$ REQUEST messages from other sites while waiting for its REPLY messages. Since a site sends REQUEST messages only in response to REQUEST messages of higher priority, on average it will send $(n-1)/4$ REQUEST messages while waiting for REPLY messages. Therefore, the average number of messages exchanged per CS execution in high demand will be $2^*[(n-1)/2+(n-1)/4] = 3^*(n-1)/2$.

9.5.4 Adaptivity in heterogeneous traffic patterns

An interesting feature of the algorithm is that its information-structure adapts itself to the environments of heterogeneous traffic of CS requests and to statistical fluctuations in traffic of CS requests to optimize the performance (the number of messages exchanged per CS execution). In non-uniform traffic environments, sites with higher traffic of CS requests will position themselves towards the right end of the staircase pattern. That is, sites with higher traffic of CS requests will tend to have lower cardinality of their request sets. Also, at a high traffic site S_i, if $S_j \in R_i$, then S_j is also a high traffic site (this comes intuitively because all high traffic sites will cluster towards the right end of the staircase). Consequently, high traffic sites will mostly send REQUEST messages only to other high traffic sites and will seldom send REQUEST messages to sites with low traffic. This adaptivity results in a reduction in the number of messages as well as a delay in granting CS in environments of heterogeneous traffic.

9.6 Lodha and Kshemkalyani's fair mutual exclusion algorithm

Lodha and Kshemakalyani's algorithm [13] (Algorithm 9.4) decreases the message complexity of the Ricart–Agrawala algorithm by using the following

interesting observation: when a site is waiting to execute the CS, it need not receive REPLY messages from every other site. To enter the CS, a site only needs to receive a REPLY message from the site whose request just precedes its request in priority. For example, if sites S_{i_1},S_{i_2}, ..S_{i_n} have a pending request for CS and the request of S_{i_1} has the highest priority and that of S_{i_n} has the lowest priority and the priority of requests decreases from S_{i_1} to S_{i_n}, then a site S_{i_k} only needs a REPLY message from site $S_{i_{k-1}}$, $1 < k \leq n$ to enter the CS.

9.6.1 System model

Each request is assigned a priority *ReqID* and requests for CS access are granted in the order of decreasing priority. We will defer the details of what *ReqID* is composed of to later sections. The underlying communication network is assumed to be error free.

Definition 9.1 *R_i and R_j are concurrent iff P_i's REQUEST message is received by P_j after P_j has made its request and P_j's REQUEST message is received by P_i after P_i has made its request.*

Definition 9.2 *Given R_i, we define the concurrency set of R_i as follows:* $\text{CSet}_i = \{R_j \mid R_i \text{ is concurrent with } R_j\} \bigcup \{R_i\}$.

9.6.2 Description of the algorithm

Algorithm 9.4 uses three types of messages (REQUEST, REPLY, and FLUSH) and obtains savings on the number of messages exchanged per CS access by assigning multiple purposes to each. For the purpose of blocking a mutual exclusion request, every site S_i has a data structure called *local_request_queue* (denoted as LRQ_i), which contains all concurrent requests made with respect to S_i's request, and these requests are ordered with respect to their priority.

All requests are totally ordered by their priorities and the priority is determined by the timestamp of the request. Hence, when a process receives a REQUEST message from some other process, it can immediately determine if it is allowed to access the CS before the requesting process or after it.

In this algorithm, messages play multiple roles and this will be discussed first.

Multiple uses of a REPLY message

1. A REPLY message acts as a reply from a process that is not requesting.
2. A REPLY message acts as a collective reply from processes that have higher priority requests.

A REPLY(R_j) from a process P_j indicates that R_j is the request made by P_j for which it has executed the CS. It also indicates that all the requests with priority $\geq$ priority of R_j have finished executing CS and are no longer in contention.

Thus, in such situations, a REPLY message is a logical reply and denotes a collective reply from all processes that had made higher priority requests.

Uses of a FLUSH message

Similar to a REPLY message, a FLUSH message is a logical reply and denotes a collective reply from all processes that had made higher priority requests. After a process has exited the CS, it sends a FLUSH message to a process requesting with the next highest priority, which is determined by looking up the process's local request queue. When a process P_i finishes executing the CS, it may find a process P_j in one of the following states:

1. R_j is in the local queue of P_i and located in some position after R_i, which implies that R_j is concurrent with R_i.
2. P_j had replied to R_i and P_j is now requesting with a lower priority. (Note that in this case R_i and R_j are not concurrent.)
3. P_j's requst had higher priority than P_i's (implying that it had finished the execution of the CS) and is now requesting with a lower priority. (Note that in this case R_i and R_j are not concurrent.)

A process P_i, after executing the CS, sends a FLUSH message to a process identified in state 1 above, which has the next highest priority, whereas it sends REPLY messages to the processes identified in states 2 and 3 as their requests are not concurrent with R_i (the resuests of processes in states 2 and 3 were deferred by P_i till it exits the CS). Now it is up to the process receiving the FLUSH message and the processes recieving REPLY messages in states 2 and 3 to determine who is allowed to enter the CS next.

Consider a scenario where we have a set of requests R_3, R_0, R_2, R_4, R_1 ordered in decreasing priority, where R_0, R_2, R_4 are concurrent with one another, then P_0 maintains a local queue of [R_0, R_2, R_4] and, when it exits the CS, it sends a FLUSH (only) to P_2.

Multiple uses of a REQUEST message

Considering two processes P_i and P_j, there can be two cases:

Case 1 P_i and P_j are not concurrently requesting. In this case, the process which requests first will get a REPLY message from the other process.

Case 2 P_i and P_j are concurrently requesting. In this case, there can be two subcases:

1. P_i is requesting with a higher priority than P_j. In this case, P_j's REQUEST message serves as an implicit REPLY message to P_i's request. Also, P_j should wait for REPLY/FLUSH message from some process to enter the CS.
2. P_i is requesting with a lower priority than P_j. In this case, P_i's REQUEST message serves as an implicit REPLY message to P_j's request. Also, P_i should wait for REPLY/FLUSH message from some process to enter the CS.

(1) *Initial local state for process* P_i:
- **int** $My\_Sequence\_Number_i = 0$
- **array of boolean** $RV_i[j] = 0, \forall j \in \{1...N\}$
- **queue of ReqID** LRQ_i is NULL
- **int** $Highest\_Sequence\_Number\_Seen_i = 0$

(2) *InvMutEx*: Process P_i executes the following to invoke mutual exclusion:
(2a) $My\_Sequence\_Number_i = Highest\_Sequence\_Number\_Seen_i + 1$.
(2b) $LRQ_i = NULL$.
(2c) Make REQUEST(R_i) message, where $R_i = (My\_Sequence\_Number_i, i)$.
(2d) Insert this REQUEST in LRQ_i in sorted order.
(2e) Send this REQUEST message to all other processes.
(2f) $RV_i[k] = 0 \forall k \in \{1 \ldots N\} - \{i\}$. $RV_i[i]$=1.

(3) *RcvReq*: Process P_i receives REQUEST(R_j), where $R_j = (SN, j)$, from process P_j:
(3a) $Highest\_Sequence\_Number\_Seen_i = max(Highest\_Sequence\_Number\_Seen_i, SN)$.
(3b) If P_i is requesting:
(3bi) If $RV_i[j] = 0$, then insert this request in LRQ_i (in sorted order) and mark $RV_i[j] = 1$. If (*CheckExecuteCS*), then execute CS.
(3bii) If $RV_i[j] = 1$, then defer the processing of this request, which will be processed after P_i executes CS.
(3c) If P_i is not requesting, then send a REPLY(R_i) message to P_j. R_i denotes the ReqID of the last request made by P_i that was satisfied.

(4) *RcvReply*: Process P_i receives REPLY(R_j) message from process P_j. R_j denotes the ReqID of the last request made by P_j that was satisfied:
(4a) $RV_i[j]$=1.
(4b) Remove all requests from LRQ_i that have a priority $\geq$ the priority of R_j.
(4c) If (*CheckExecuteCS*), then execute CS.

(5) *FinCS*: Process P_i finishes executing CS:
(5a) Send FLUSH(R_i) message to the next candidate in LRQ_i. R_i denotes the ReqID that was satisfied.
(5b) Send REPLY(R_i) to the deferred requests. R_i is the ReqID corresponding to which P_i just executed the CS.

(6) *RcvFlush*: Process P_i receives a FLUSH(R_j) message from a process P_j:
(6a) $RV_i[j] = 1$
(6b) Remove all requests in LRQ_i that have the priority $\geq$ the priority of R_j.
(6c) If (*CheckExecuteCS*) then execute CS.

(7) *CheckExecuteCS*: If ($RV_i[k] = 1, \forall k \in \{1 \ldots N\}$) and P_i's request is at the head of LRQ_i, then return *true*, else return *false*.

Algorithm 9.4 Lodha and Kshemkalyani's fair mutual exclusion algorithm [13].

Examples

- **Figure 9.11** Processes P_1 and P_2 are concurrent and they send out REQUESTs to all other processes. The REQUEST sent by P_1 to P_3 is delayed and hence is not shown until in Figure 9.13.
- **Figure 9.12** When P_3 receives the REQUEST from P_2, it sends REPLY to P_2.
- **Figure 9.13** The delayed REQUEST of P_1 arrives at P_3 and at the same time, P_3 sends out its REQUEST for CS, which makes it concurrent with the request of P_1.
- **Figure 9.14** P_1 exits the CS and sends out a FLUSH message to P_2.
- **Figure 9.15** Since the requests of P_2 and P_3 are not concurrent, P_2 sends a FLUSH message to P_3. P_3 removes (1,1) from its local queue and enters the CS.

The data structures LRQ and RV are updated in each step as discussed previously.

9.6.3 Safety, fairness and liveness

Proofs for safety, fairness and liveness are quite involved and interested readers are referred to the original paper for detailed proofs.

9.6.4 Message complexity

To execute the CS, a process P_i sends $(N-1)$ REQUEST messages. It receives $(N-|\ CSet_i\ |)$ REPLY messages. There are two cases to consider:

1. $|\ CSet_i\ | \geq 2$. There are two subcases here:

   (a) There is at least one request in CSet_i whose priority is smaller than that of R_i. So P_i will send one FLUSH message. In this case the total number of messages for CS access is $2N-|\ CSet_i\ |$. When all the requests are concurrent, this reduces to N messages.

   (b) There is no request in $CSet_i$, whose priority is less than the priority of R_i. P_i will not send a FLUSH message. In this case, the total number of messages for CS access is $2N-1-|\ CSet_i\ |$. When all the requests are concurrent, this reduces to $N-1$ messages.

2. $|\ CSet_i\ | = 1$. This is the worst case, implying that all requests are satisfied serially. P_i will not send a FLUSH message. In this case, the total number of messages for CS access is $2(N-1)$ messages.

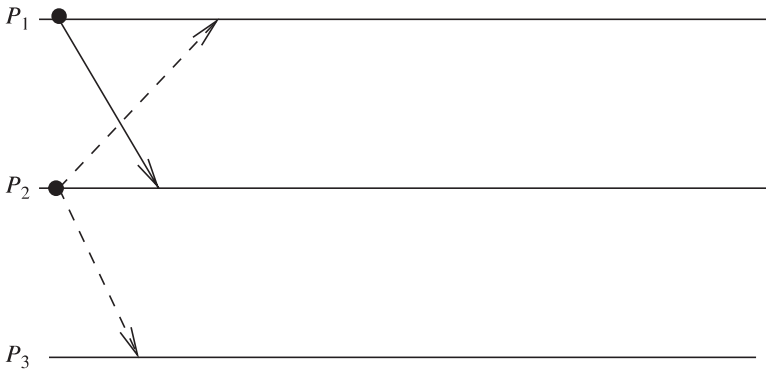

Figure 9.11 Processes P_1 and P_2 send out REQUESTs.

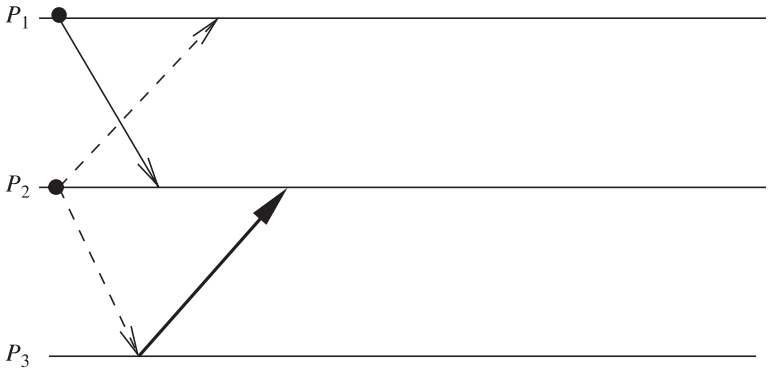

Figure 9.12 P_3 sends a REPLY message to P_2 only.

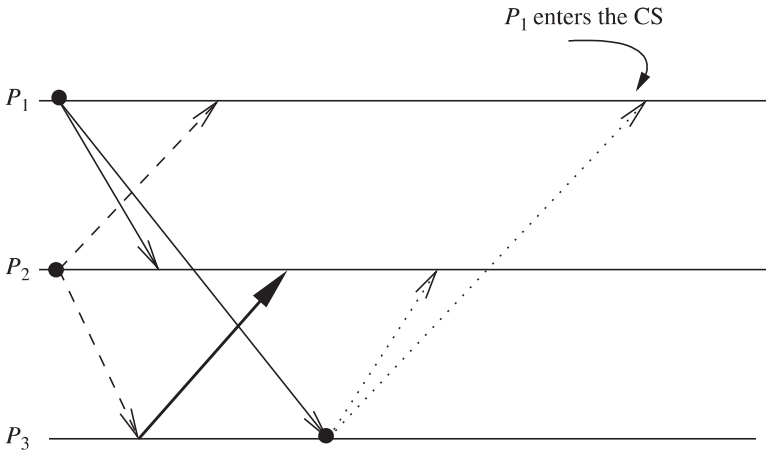

Figure 9.13 P_3 sends out a REQUEST message.

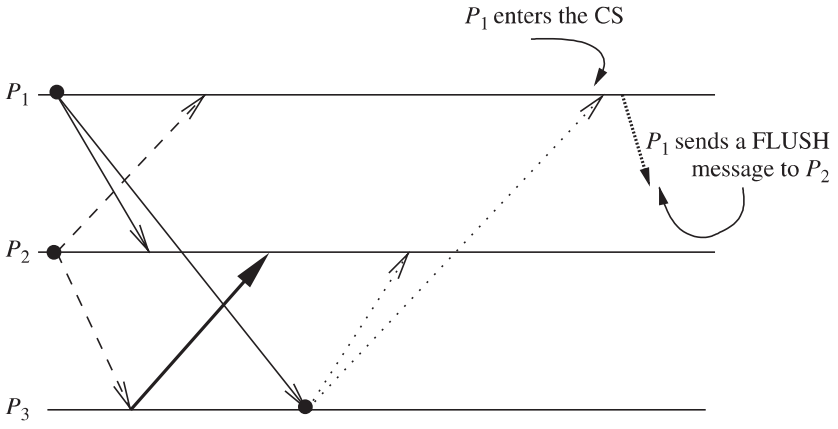

Figure 9.14 P_1 exits the CS and sends a FLUSH message to P_2.

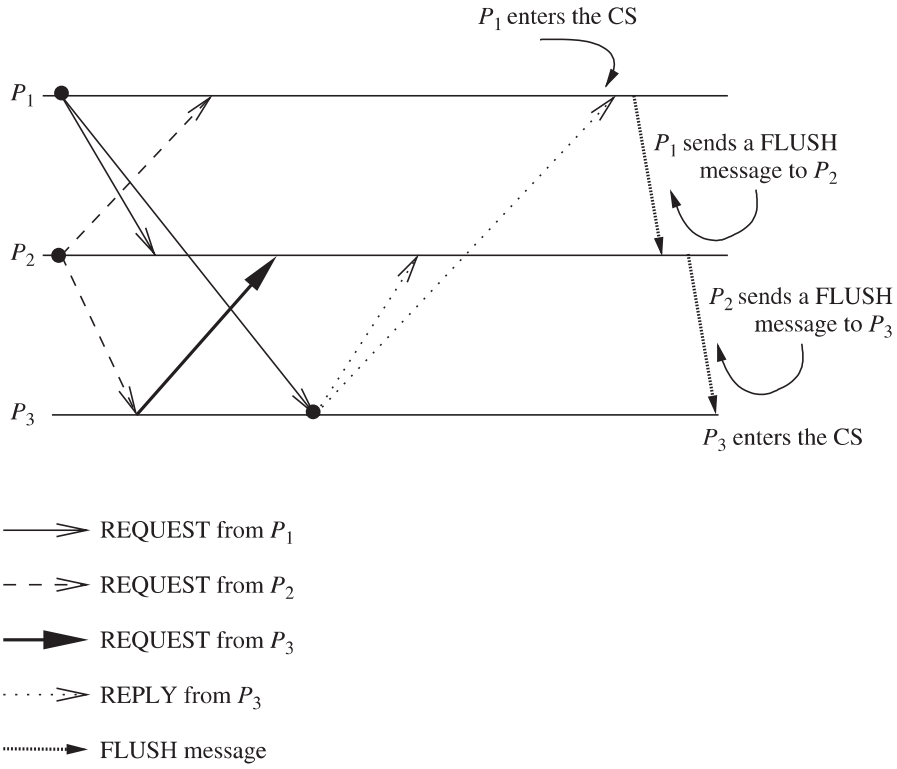

Figure 9.15 P_3 enters the CS.

9.7 Quorum-based mutual exclusion algorithms

Quorum-based mutual exclusion algorithms respresented a departure from the trend in the following two ways:

1. A site does not request permission from all other sites, but only from a subset of the sites. This is a radically different approach as compared to the Lamport and Ricart–Agrawala algorithms, where all sites participate in conflict resolution of all other sites. In quorum-based mutual exclusion algorithm, the request set of sites are chosen such that $\forall i \; \forall j : 1 \leq i, j \leq N :: R_i \cap R_j \neq \Phi$. Consequently, every pair of sites has a site which mediates conflicts between that pair.
2. In quorum-based mutual exclusion algorithm, 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. Therefore, a site S_i *locks* all the sites in R_i in exclusive mode before executing its CS.

Quorum-based mutual exclusion algorithms significantly reduce the message complexity of invoking mutual exclusion by having sites ask permission from only a subset of sites.

Since these algorithms are based on the notion of "Coteries" and "Quorums," we first describe the idea 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 the first and third sets do not have a common element.
- **Minimality property** There should be no quorums g, h in coterie C such that $g \supseteq h$. 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.

Coteries and quorums can be used to develop algorithms to ensure mutual exclusion in a distributed environment. A simple protocol works as follows: let "a" be a site in quorum "A." If "a" wants to invoke mutual exclusion, it requests permission from all sites in its quorum "A." Every site does the same to invoke mutual exclusion. Due to the Intersection property, quorum "A" contains at least one site that is common to the quorum of every other site. These common sites send permission to only one site at any time. Thus, mutual exclusion is guaranteed.

Note that the Minimality property ensures efficiency rather than correctness. In the simplest form, quorums are formed as sets that contain a majority of sites. There exists a variety of quorums and a variety of ways to construct quorums. For example, Maekawa [14] used the theory of projective planes to develop quorums of size $\sqrt{N}$.

9.8 Maekawa's algorithm

Maekawa's algorithm [14] 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 $|R_i| = \sqrt{N}$.

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 every pair of sites.

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, which implies 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.

In Maekawa's algorithm, a site S_i executes the steps shown in Algorithm 9.5 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.

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.

Algorithm 9.5 Maekawa's algorithm.

Correctness

Theorem 9.3 *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

Note that the size of a request set is $\sqrt{N}$. Therefore, 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. 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.

9.8.1 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 loss of generality, 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 $R_k \cap R_i = \{S_{ki}\}$. 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: S_{ij} has been locked by S_i (forcing S_j to wait at S_{ij}), S_{jk} has been locked by S_j (forcing S_k to wait at S_{jk}), and 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 (unless the former has succeeded in acquiring locks on all the needed sites) [14, 22]. 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.

Deadlock handling requires the following three types of messages:

FAILED A FAILED message from site S_i to site S_j indicates that S_i cannot 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).

Details of how Maekawa's algorithm handles deadlocks are 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.

Thus, Maekawa-type algorithms require extra messages to handle deadlocks and may exchange these messages even though there is no deadlock. The maximum number of messages required per CS execution in this case is $5\sqrt{N}$.

9.9 Agarwal–El Abbadi quorum-based algorithm

Agarwal and El Abbadi [1] developed a simple and efficient mutual exclusion algorithm by introducing tree quorums. They gave a novel algorithm for constructing tree-structured quorums in the sense that it uses hierarchical structure of a network. The mutual exclusion algorithm is independent of the underlying topology of the network and there is no need for a multicast facility in the network. However, such facility will improve the performance of the algorithm. The mutual exclusion algorithm assumes that sites in the distributed system can be organized into a structure such as tree, grid, binary tree, etc. and there exists a routing mechanism to exchange messages between different sites in the system.

The Agarwal–El Abbadi quorum-based algorithm, however, constructs quorums from trees. Such quorums are called "tree-structured quorums." The following sections describe an algorithm for constructing tree-structured quorums and present an analysis of the algorithm and a protocol for mutual exclusion in distributed systems using tree-structured quorums.

9.9.1 Constructing a tree-structured quorum

All the sites in the system are logically organized into a complete binary tree. To build such a tree, any site could be chosen as the root, any other two sites may be chosen as its children, and so on. For a complete binary tree with

level "k," we have $2^{k+1}-1$ sites with its root at level k and leaves at level 0. The number of sites in a path from the root to a leaf is equal to the level of the tree $k+1$, which is equal to $O(\log n)$. There will be 2^k leaves in the tree. A path in a binary tree is the sequence $a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_k$ such that a_i is the parent of a_{i+1}.

The algorithm for constructing structured quorums from the tree is given in Algorithm 9.6. For the purpose of presentation, we assume that the tree is complete, however, the algorithm works for any arbitrary binary tree.

```
(1)  FUNCTION GetQuorum(Tree: NetworkHierarchy): QuorumSet;
(2)    VAR left, right: QuorumSet;
(3)      BEGIN
(4)      IF Empty (Tree) THEN
(5)           RETURN ({});
(6)      ELSE IF GrantsPermission(Tree↑.Node) THEN
(7)             RETURN((Tree↑.Node) ∪ GetQuorum (Tree↑.LeftChild));
(8)             OR
(9)             RETURN((Tree↑.Node) ∪ GetQuorum (Tree↑.RightChild));
(10)          ELSE
(11)              left ← GetQuorum(Tree↑.left);
(12)              right ← GetQuorum(Tree↑.right);
(13)              IF (left = ∅ ∨ right = ∅) THEN
(14)                   (* Unsuccessful in establishing a quorum *)
(15)                    EXIT(-1);
(16)              ELSE
(17)                   RETURN(left ∪ right);
(18)              END; (* IF *)
(19)          END; (* IF *)
(20)     END; (* IF *)
(21)  END GetQuorum
```

Algorithm 9.6 Algorithm for constructing a tree-structured quorum [1].

The algorithm for constructing tree-structured quorums uses two functions called *GetQuorum*(*Tree*) and *GrantsPermission*(*site*) and assumes that there is a well-defined root for the tree. *GetQuorum* is a recursive function that takes a tree node "x" as the parameter and calls *GetQuorum* for its child node provided that the *GrantsPermission*(x) is true. The *GrantsPermission*(x) is true only when the node "x" agrees to be in the quorum. If the node "x" is down due to a failure, then it may not agree to be in the quorum and the value of *GrantsPermission*(x) will be false. The algorithm tries to construct quorums in a way that each quorum represents any path from the root to a leaf, i.e., in this case the (no failures) quorum is any set $a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_k$, where a_1 is the root and a_k is a leaf, and for all $i < k$, a_i is the parent of a_{i+1}. If it fails to find such a path (say, because node "x" has failed), the control goes to the ELSE block which specifies that the failed node "x"

is substituted by two paths both of which start with the left and right children of "x" and end at leaf nodes. Note that each path must terminate in a leaf site. If the leaf site is down or inaccessible due to any reason, then the quorum cannot be formed and the algorithm terminates with an error condition. The sets that are constructed using this algorithm are termed as *tree quorums*.

9.9.2 Analysis of the algorithm for constructing tree-structured quorums

The best case scenario of the algorithm takes $O(\log\ n)$ sites to form a tree quorum. There are certain cases where even in the event of a failure, $O(\log\ n)$ sites are sufficient to form a tree quorum. For example, if the site that is parent of a leaf node fails, then the number of sites that are necessary for a quorum will be still $O(\log\ n)$. Thus, the algorithm requires very few messages in a relatively fault-free environment. It can tolerate the failure up to $n - \mathrm{O}(\log\ n)$ sites and still form a tree quorum. In the worst case, the algorithm requires the majority of sites to construct a tree quorum and the number of sites is same for all cases (faults or no faults). The worst case tree quorum size is determined as $O((n+1)/2)$ by induction.

9.9.3 Validation

The tree quorums constructed by the above algorithm are valid, i.e., they conform to the coterie properties such as Intersection property and Minimality property. To prove the correctness of the algorithm, consider a binary tree with level $k+1$. Assume that root of the tree is a_1. The tree can be viewed as consisting of a root, a left subtree, and a right subtree. According to Algorithm 9.6, the constructed quorums contain one of the following:

1. $\{a_1\} \cup \{$ sites from the left subtree$\}$;
2. $\{a_1\} \cup \{$sites from the right subtree$\}$;
3. {sites from the quorum set of left subtree} $\cup$ {sites from the quorum set of right subtree}.

Clearly, the quorum of type 1 has non-empty intersection with those quorums formed using types 2 or 3, which shows that the Intersection property holds true. Also, the members in the quorum of type 1 are not contained in quorums of types 2 and 3. Thus, the Minimality property holds true. Similar conditions exist for quorums of types 2 and 3. This forms as the basis for proving correctness of the algorithm based on induction.

9.9.4 Examples of tree-structured quorums

Now we present examples of tree-structured quorums for a better understanding of the algorithm. In the simplest case, when there is no node failure, the number of quorums formed is equal to the number of leaf sites.

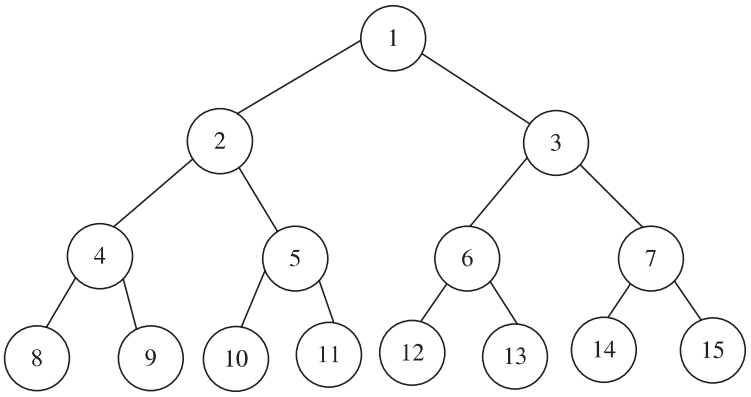

Figure 9.16 A tree of 15 sites.

Consider the tree of height 3 shown in Figure 9.16 constructed from 15 ($2^{3+1} - 1$) sites. Now, a quorum has all sites along any path from root to leaf. In this case eight quorums are formed from eight possible root-leaf paths: 1–2–4–8, 1–2–4–9, 1–2–5–10, 1–2–5–11, 1–3–6–12, 1–3–6–13, 1–3–7–14 and 1–3–7–15. If any site fails, the algorithm substitutes for that site two possible paths starting from the site's two children and ending in leaf nodes. For example, when node 3 fails, we consider the possible paths starting from children 6 and 7 and ending at the leaf nodes. The possible paths starting from child 6 are 6–12 and 6–13, while the possible paths starting from child 7 are 7–14 and 7–15. So, when node 3 fails, the following eight quorums can be formed: {1,6,12,7,14}, {1,6,12,7,15}, {1,6,13,7,14}, {1,6,13,7,15}, {1,2,4,8}, {1,2,4,9}, {1,2,5,10}, {1,2,5,11}.

If a failed site is a leaf node, the operation has to be aborted and a tree-structured quorum cannot be formed (see lines 13–15 of the algorithm above). However, quorum formation can continue with other working nodes. Since the number of nodes from root to leaf in an "n" node complete tree is log n, the best case for quorum formation, i.e, the least number of nodes needed for a quorum is log n. In the worst case, a majority of sites are needed for mutual exclusion. For example, if sites 1 and 2 are down in Figure 9.16, the quorums that are formed must include either {4,8} or {4,9} and either{5,10} or {5,11} and one of the four paths {3,6,12}, {3,6,13} {3,7,14} or {3,7,15}. In this case, the following are the candidates for quorums: {4,5,3,6,8,10,12}, {4,5,3,6,8,10,13}, {4,5,3,6,8,11,12}, {4,5,3,6,8,11,13}, {4,5,3,6,9,10,12}, {4,5,3,6,9,10,13}, {4,5,3,6,9,11,12}, {4,5,3,6,9,11,13}, {4,5,3,7,8,10,14}, {4,5,3,7,8,10,15}, {4,5,3,7,8,11,14}, {4,5,3,7,8,11,15}, {4,5,3,7,9,10,14}, {4,5,3,7,9,10,15}, {4,5,3,7,9,11,14}, and {4,5,3,7,9,11,15}.

When the number of node failures is greater than or equal to *log n*, the algorithm may not be able to form tree-structured quorum. For example when sites 1, 2, 4, and 8 are inaccessible, the set of sites {3,5,6,7,8,9,10,11,12,13,14,15} form a majority of sites but not a structured quorum. So, as long as the

number of site failures is less than log n, the tree quorum algorithm gurantees the formation of a quorum and it exhibits the property of "graceful degradation," which is useful in distributed fault tolerance. As failures occur and increase, the probability of forming quorums decreases and mutual exclusion is achieved at increasing costs because when a node fails, instead of one path from node, the quorum must include two paths starting from the node's children. For example, in a tree of level k, the size of quorum is $(k+1)$. If a node failure occurs at level $i > 0$, then the quorum size increases to $(k-i)+2i$. The penalty is severe when the failed node is near the root. Thus, the tree quorum algorithm may still allow quorums to be formed even after the failures of $n-\lfloor \log n \rfloor$ sites.

9.9.5 The algorithm for distributed mutual exclusion

We now describe the algorithm for achieving distributed mutual exclusion using tree-structured quorums. Suppose a site s wants to enter the critical section (CS). The following events should occur in the order given:

1. Site s sends a "Request" message to all other sites in the structured quorum it belongs to.
2. Each site in the quorum stores incoming requests in a *request queue*, ordered by their timestamps.
3. A site sends a "Reply" message, indicating its consent to enter CS, only to the request at the head of its *request queue*, having the lowest timestamp.
4. If the site s gets a "Reply" message from all sites in the structured quorum it belongs to, it enters the CS.
5. After exiting the CS, s sends a "Relinquish" message to all sites in the structured quorum. On the receipt of the "Relinquish" message, each site removes s's request from the head of its *request queue*.
6. If a new request arrives with a timestamp smaller than the request at the head of the queue, an "Inquire" message is sent to the process whose request is at the head of the queue and waits for a "Yield" or "Relinquish" message.
7. When a site s receives an "Inquire" message, it acts as follows:
   - If s has acquired all of its necessary replies to access the CS, then it simply ignores the "Inquire" message and proceeds normally and sends a "Relinquish" message after exiting the CS.
   - If s has not yet collected enough replies from its quorum, then it sends a "Yield" message to the inquiring site.
8. When a site gets the "Yield" message, it puts the pending request (on behalf of which the "Inquire" message was sent) at the head of the queue and sends a "Reply" message to the requestor.

9.9.6 Correctness proof

Mutual exclusion is guaranteed because the set of quorums satisfy the Intersection property. Proof for freedom from deadlock is similar to that of Maekawa's algorithm. The readers are referred to the original source [14].

Example Consider a coterie C which consists of quorums {1,2,3}, {2,4,5}, and {4,1,6}. Suppose nodes 3, 5, and 6 want to enter CS, and they send requests to sites (1, 2), (2, 4), and (1, 4), respectively. Suppose site 3's request arrives at site 2 before site 5's request. In this case, site 2 will grant permission to site 3's request and reject site 5's request. Similarly, suppose site 3's request arrives at site 1 before site 6's request. So site 1 will grant permission to site 3's request and reject site 6's request. Since sites 5 and 6 did not get consent from all sites in their quorums, they do not enter the CS. Since site 3 alone gets consent from all sites in its quorum, it enters the CS and mutual exclusion is achieved.

9.10 Token-based algorithms

In token-based algorithms, a unique token is shared among the sites. A site is allowed to enter its CS if it possesses the token. A site holding the token can enter its CS repeatedly until it sends the token to some other site. Depending upon the way a site carries out the search for the token, there are numerous token-based algorithms. Next, we discuss two token-based mutual exclusion algorithms.

Before we start with the discussion of token-based algorithms, two comments are in order. First, token-based algorithms use sequence numbers instead of timestamps. Every request for the token contains a sequence number and the sequence numbers of sites advance independently. A site increments its sequence number counter every time it makes a request for the token. (A primary function of the sequence numbers is to distinguish between old and current requests.) Second, the correctness proof of token-based algorithms, that they enforce mutual exclusion, is trivial because an algorithm guarantees mutual exclusion so long as a site holds the token during the execution of the CS. Instead, the issues of freedom from starvation, freedom from deadlock, and detection of the token loss and its regeneration become more prominent.

9.11 Suzuki–Kasami's broadcast algorithm

In Suzuki–Kasami's algorithm [29] (Algorithm 9.7), 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 that 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.

Although the basic idea underlying this algorithm may sound rather simple, there are two design issues that 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 cannot 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, but it 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 outdateded.
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 S_i receives a token request message from a site S_j, site S_j may have an outstanding request for the CS. However, after the corresponding request for the CS has been satisfied at S_j, an issue is how to inform site S_i (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 S_j has the form REQUEST(j, n) where n ($n = 1, 2, \ldots$) is a sequence number that indicates that site S_j is requesting its nth CS execution. A site S_i keeps an array of integers $RN_i[1, \ldots, N]$ where $RN_i[j]$ denotes the largest sequence number received in a REQUEST message so far from site S_j. When site S_i receives a REQUEST(j, n) message, it sets $RN_i[j] := max(RN_i[j], n)$. Thus, when a site S_i receives a REQUEST(j, n) message, the request is outdated if $RN_i[j] > n$.

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

Requesting the critical section:

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

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

Executing the critical section:

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

Releasing the critical section: Having finished the execution of the CS, site S_i takes the following actions:

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

(e) For every site S_j whose i.d. is not in the token queue, it appends its i.d. to the token queue if $RN_i[j] = LN[j] + 1$.

(f) If the token queue is nonempty after the above update, S_i deletes the top site i.d. from the token queue and sends the token to the site indicated by the i.d.

Algorithm 9.7 Suzuki–Kasami's broadcast algorithm.

Thus, as shown in Algorithm 9.7, 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 9.3 *A requesting site enters the CS in finite time.*

Proof Token request messages of a site S_i reach other sites in finite time. Since one of these sites will have token in finite time, site S_i'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 S_i will get the token and execute the CS in finite time. □

Performance

The beauty of the 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. The synchronization delay in this algorithm is 0 or T.

9.12 Raymond's tree-based algorithm

Raymond's tree-based mutual exclusion algorithm [19] uses a spanning tree of the computer network to reduce the number of messages exchanged per critical section execution. The algorithm exchanges only $O(\log N)$ messages under light load, and approximately four messages under heavy load to execute the CS, where N is the number of nodes in the network.

The algorithm assumes that the underlying network guarantees message delivery. The time or order of message arrival cannot be predicted. All nodes of the network are completely reliable. (Only for the initial part of the discussion, i.e., until node failure is discussed.) If the network is viewed as a graph, where the nodes in the network are the vertices of the graph, and the links between nodes are the edges of the graph, a spanning tree of a network of N nodes will be a tree that contains all N nodes. A minimal spanning tree is one such tree with minimum cost. Typically, this cost function is based on the network link characteristics. The algorithm operates on a minimal spanning tree of the network topology or logical structure imposed on the network.

The algorithm considers the network nodes to be arranged in an unrooted tree structure as shown in Figure 9.17. Messages between nodes traverse along the undirected edges of the tree in the Figure 9.17. The tree is also a spanning tree of the seven nodes A, B, C, D, E, F, and G. It also turns out to be a minimal spanning tree because it is the only spanning tree of these seven nodes. A node needs to hold information about and communicate only to its immediate-neighboring nodes. In Figure 9.17, for example, node C holds information about and communicates only to nodes B, D, and G; it does not need to know about the other nodes A, E, and F for the operation of the algorithm.

Similar to the concept of tokens used in token-based algorithms, this algorithm uses a concept of privilege to signify which node has the privilege to enter the critical section. Only one node can be in possession of the privilege (called the privileged node) at any time, except when the privilege is in transit

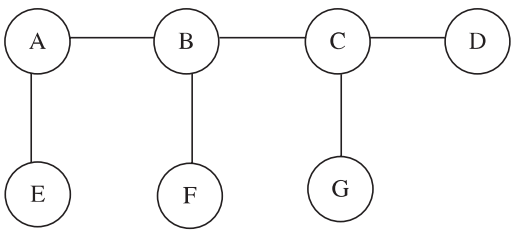

Figure 9.17 Nodes with an unrooted tree structure.

from one node to another in the form of a PRIVILEGE message. When there are no nodes requesting for the privilege, it remains in possession of the node that last used it.

9.12.1 The HOLDER variables

Each node maintains a HOLDER variable that provides information about the placement of the privilege in relation to the node itself. A node stores in its HOLDER variable the identity of a node that it thinks has the privilege or leads to the node having the privilege. The HOLDER variables of all the nodes maintain directed paths from each node to the node in the possession of the privilege.

For two nodes X and Y, if $\text{HOLDER}_X = \text{Y}$, we could redraw the undirected edge between the nodes X and Y as a directed edge from X to Y. Thus, for instance, if node G holds the privilege, Figure 9.17 can be redrawn with logically directed edges as shown in Figure 9.18. The shaded node represents the privileged node. The following will be the values of the HOLDER variables of various nodes:

$\text{HOLDER}_A = \text{B}$(Since the privilege is located in a sub-tree of A denoted by B.)

Proceeding with similar reasoning, we have

$$\text{HOLDER}_B = \text{C},$$
$$\text{HOLDER}_C = \text{G},$$
$$\text{HOLDER}_D = \text{C},$$
$$\text{HOLDER}_E = \text{A},$$
$$\text{HOLDER}_F = \text{B},$$
$$\text{HOLDER}_G = \text{self}.$$

Now suppose that node B, which does not hold the privilege, wants to execute the critical section. Then B sends a REQUEST message to HOLDER_B, i.e., C, which in turn forwards the REQUEST message to HOLDER_C, i.e., G. So a series of REQUEST messages flow between the node making the request for the privilege and the node having the privilege.

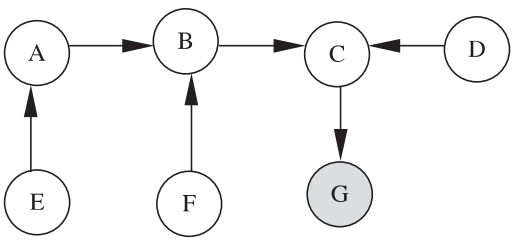

Figure 9.18 Tree with logically directed edges, all pointing in a direction towards node G – the privileged node.

Table 9.1 Variables used in the algorithm.

Variable name	Possible values	Comments
HOLDER	"self" or the identity of one of the immediate neighbors.	Indicates the location of the privileged node in relation to the current node.
USING	True or false.	Indicates if the current node is executing the critical section.
REQUEST_Q	A FIFO queue that could contain "self" or the identities of immediate neighbors as elements.	The REQUEST_Q of a node consists of the identities of those immediate neighbors that have requested for privilege but have not yet been sent the privilege.
ASKED	True or false.	Indicates if node has sent a request for the privilege.

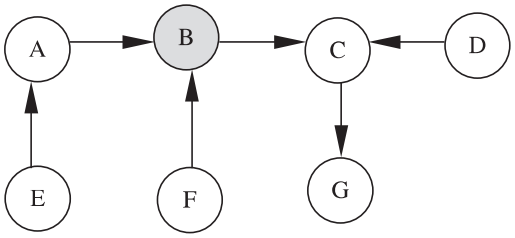

Figure 9.19 Tree with logically directed edges, all pointing in a direction towards node G – the privileged node.

The privileged node G, if it no longer needs the privilege, sends the PRIVILEGE message to its neighbor C, which made a request for the privilege, and resets HOLDER_G to C. Node C, in turn, forwards the PRIVILEGE to node B, since it had requested the privilege on behalf of B. Node C also resets HOLDER_C to B. The tree in Figure 9.18 will now look as shown in Figure 9.19.

Thus, at any stage, except when the PRIVILEGE message is in transit, the HOLDER variables collectively make sure that directed paths are maintained from each of the $N - 1$ nodes to the privileged node in the network.

9.12.2 The operation of the algorithm

Data structures

Each node maintains variables that are defined in Table 9.1. The value "self" is placed in REQUEST_Q if the node makes a request for the privilege for its own use. The maximum size of REQUEST_Q of a node is the number of immediate neighbors $+1$ (for "self"). ASKED prevents the sending of duplicate requests for privilege, and also makes sure that the REQUEST_Qs of the various nodes do not contain any duplicate elements.

9.12.3 Description of the algorithm

The algorithm consists of the following parts:

- ASSIGN_PRIVILEGE;
- MAKE_REQUEST;
- events;
- message overtaking.

ASSIGN_PRIVILEGE

This is a routine to effect the sending of a PRIVILEGE message. A privileged node will send a PRIVILEGE message if:

- it holds the privilege but is not using it;
- its REQUEST_Q is not empty; and
- the element at the head of its REQUEST_Q is not "self." That is, the oldest request for privilege must have come from another node.

A situation where "self" is at the head of REQUEST_Q may occur immediately after a node receives a PRIVILEGE message. The node will enter into the critical section after removing "self" from the head of REQUEST_Q. If the i.d. of another node is at the head of REQUEST_Q, then it is removed from the queue and a PRIVILEGE message is sent to that node. Also, the variable ASKED is set to false since the currently privileged node will not have sent a request to the node (called HOLDER-to-be) that is about to receive the PRIVILEGE message.

MAKE_REQUEST

This is a routine to effect the sending of a REQUEST message. An unprivileged node will send a REQUEST message if:

- it does not hold the privilege;
- its REQUEST_Q is not empty, i.e., it requires the privilege for itself, or on behalf of one of its immediate neighboring nodes; and
- it has not sent a REQUEST message already.

The variable ASKED is set to true to reflect the sending of the REQUEST message. The MAKE_REQUEST routine makes no change to any other variables. The variable ASKED will be true at a node when it has sent REQUEST message to an immediate neighbor and has not received a response. The variable will be false otherwise. A node does not send any REQUEST messages, if ASKED is true at that node. Thus the variable ASKED makes sure that unnecessary REQUEST messages are not sent from the unprivileged node, and consequently ensures that the REQUEST_Q of an immediate neighbor does not contain duplicate entries of a neighboring node. This makes the REQUEST_Q of any node bounded, even when operating under heavy load.

Table 9.2 Events in the algorithms.

Event	Algorithm functionality
A node wishes to execute critical section.	Enqueue (REQUEST_Q, Self); ASSIGN_PRIVILEGE; MAKE_REQUEST
A node receives a REQUEST message from one of its immediate neighbors X.	Enqueue(REQUEST_Q, X); ASSIGN_PRIVILEGE; MAKE_REQUEST
A node receives a PRIVILEGE message.	HOLDER := self; ASSIGN_PRIVILEGE; MAKE_REQUEST
A node exits the critical section.	USING := false; ASSIGN_PRIVILEGE; MAKE_REQUEST

Events

The four events that constitute the algorithm are shown in Table 9.2.

- **A node wishes critical section entry** If it is the privileged node, the node could enter the critical section using the ASSIGN_PRIVILEGE routine. If not, the node could send a REQUEST message using the MAKE_REQUEST routine in order to get the privilege.
- **A node receives a REQUEST message from one of its immediate neighbors** If this node is the current HOLDER, it may send the PRIVILEGE to a requesting node using the ASSIGN_PRIVILEGE routine. If not, it could forward the request using the MAKE_REQUEST routine.
- **A node receives a PRIVILEGE message** The ASSIGN_PRIVILEGE routine could result in the execution of the critical section at the node, or may forward the privilege to another node. After the privilege is forwarded, the MAKE_REQUEST routine could send a REQUEST message to reacquire the privilege, for a pending request at this node.
- **A node exits the critical section** On exit from the critical section, this node may pass the privilege on to a requesting node using the ASSIGN_PRIVILEGE routine. It may then use the MAKE_REQUEST routine to get back the privilege, for a pending request at this node.

Message overtaking

This algorithm does away with the use of sequence numbers because of its inherent operations and by the acyclic structure it employs. Figure 9.20 shows the logical pattern of message flow between any two neighboring nodes (nodes A and B here).

If any message overtaking occurs between nodes A and B, it can occur when a PRIVILEGE message is sent from node A to node B, which is then very closely followed by a REQUEST message from node A to node B. In other words, node A sends the privilege and immediately wants it back. Such

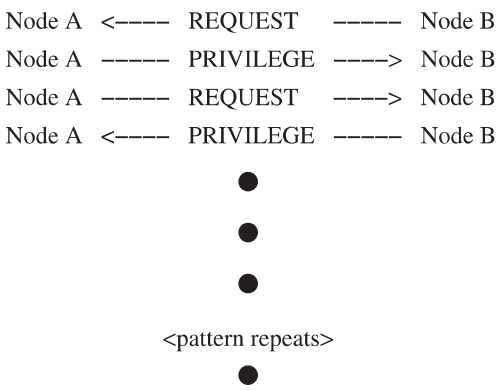

Figure 9.20 Logical pattern of message flow between neighboring nodes A and B.

message overtaking as described above will not affect the operation of the algorithm. If node B receives the REQUEST message from node A before receiving the PRIVILEGE message from node A, A's request will be queued in REQUEST_Q_B. Since B is not a privileged node, it will not be able to send a privilege to node A in reply. When node B receives the PRIVILEGE message from A after receiving the REQUEST message, it could enter the critical section or could send a PRIVILEGE message to an immediate neighbor at the head of REQUEST_Q_B, which need not be node A. So message overtaking does not affect the algorithm.

9.12.4 Correctness

The algorithm provides the following guarantees:

- mutual exclusion is guaranteed;
- deadlock is impossible;
- starvation is impossible.

Mutual exclusion is guaranteed

The algorithm ensures that, at any instant of time, no more than one node holds the privilege, which is a necessity for mutual exclusion. Whenever a node receives a PRIVILEGE message, it becomes privileged. Similarly, whenever a node sends a PRIVILEGE message, it becomes unprivileged. Between the instants one node becomes unprivileged and another node becomes privileged, there is no privileged node. Thus, there is at most one privileged node at any point of time in the network.

Deadlock is impossible

When the critical section is free, and one or more nodes want to enter the critical section but are not able to do so, a deadlock may occur. This could happen due to any of the following scenarios:

1. The privilege cannot be transferred to a node because no node holds the privilege.

2. The node in possession of the privilege is unaware that there are other nodes requiring the privilege.
3. The PRIVILEGE message does not reach the requesting unprivileged node.

None of the above three scenarios can occur in this algorithm, thus guarding against deadlocks. Scenario 1 can never occur in this algorithm because we have assumed that nodes do not fail and messages are not lost. There can never be a situation where REQUEST messages do not arrive at the privileged node. The logical pattern established using HOLDER variables ensures that a node that needs the privilege sends a REQUEST message either to a node holding the privilege or to a node that has a path to a node holding the privilege. Thus scenario 2 can never occur in this algorithm. The series of REQUEST messages are enqueued in the REQUEST_Qs of various nodes such that the REQUEST_Qs of those nodes collectively provide a logical path for the tranfer of the PRIVILEGE message from the privileged node to the requesting unprivileged nodes. So scenario 3 can never occur in this algorithm.

Starvation is impossible

When node A holds the privilege, and node B requests the privilege, the identity of B or the i.d.s of the proxy nodes for node B will be present in the REQUEST_Qs of various nodes in the path connecting the requesting node to the currently privileged node. So, depending upon the position of the i.d. of node B in those REQUEST_Qs, node B will sooner or later receive the privilege. Thus once node B's REQUEST message reaches the privileged node A, node B is sure to receive the privilege.

To better illustrate, let us consider Figure 9.19. Node B is the current holder of the privilege. Suppose that node C is already at the head of REQUEST_Q_B. Assume that the REQUEST_Qs of all other nodes are empty. Now if node E wants to enter the critical section, it will send a REQUEST message to its immediate neighbor, node A. We will show that node E does not starve. Assume that B is executing the critical section by the time E's REQUEST is propagated to node B. At this instance, the REQUEST_Qs of E, A, and B will be as follows:

$$\text{REQUEST\_Q}_E = \text{self},$$
$$\text{REQUEST\_Q}_A = \text{E},$$
$$\text{REQUEST\_Q}_B = \text{C, A}.$$

When node B exits the critical section, it removes the node at the head of REQUEST_Q_B, i.e., node C, and send the privilege to node C. Node B will then send a REQUEST to node C on behalf of node A, which requested privilege on behalf of node E. After node C receives the privilege and completes

executing the critical section, the REQUEST_Qs of nodes C, B, A, and E will look as follows:

$$\text{REQUEST\_Q}_C = \text{B},$$
$$\text{REQUEST\_Q}_B = \text{A},$$
$$\text{REQUEST\_Q}_A = \text{E},$$
$$\text{REQUEST\_Q}_E = \text{self}.$$

Now, the next node to receive the privilege will be node E, a fact that is represented by the logical path "BAE" that the REQUEST_Qs of nodes C, B, and A form. Since node B had requested privilege on behalf of node A, and node A on behalf of node E, the PRIVILEGE ultimately gets propagated to node E. Thus, a node never starves.

9.12.5 Cost and performance analysis

The algorithm exhibits the following worst-case cost: (2 * longest path length of the tree) messages per critical section entry. This happens when the privilege is to be passed between nodes at either end of the longest path of the minimal spanning tree. Thus the worst possible network topology for this algorithm will be one where all nodes are arranged in a straight line. In a straight line the longest path length will be $N - 1$, and thus the algorithm will exchange $2 * (N - 1)$ messages per CS execution. However, if all nodes generate equal number of REQUEST messages for the privilege, the average number of messages needed per critical section entry will be approximately $2N/3$ because the average distance between a requesting node and a privileged node is $(N+1)/3$.

The best topology for the algorithm is the radiating star topology. The worst-case cost of this algorithm for this topology is $O(\log_{K-1} N)$. Even among radiating star topologies, trees with higher fan-outs are preferred. The longest path length of such trees is typically $O(\log N)$. Thus, on average, this algorithm involves the exchange of $O(\log N)$ messages per critical section execution.

When under heavy load, the algorithm exhibits an interesting property: "as the number of nodes requesting the privilege increases, the number of messages exchanged per critical section entry decreases." In fact, it requires the exchange of only four messages per CS execution as explained below.

When all nodes are sending privilege requests, PRIVILEGE messages travel along all $N-1$ edges of the minimal spanning tree exactly twice to give the privilege to all N nodes. Each of these PRIVILEGE messages travel in response to a REQUEST message. Thus, a total of $4 * (N-1)$ messages travel across the minimal spanning tree. Hence, the total number of messages exchanged per critical section execution is $4(N-1)$/N, which is approximately 4.

9.12.6 Algorithm initialization

Algorithm initialization begins with one node being chosen as the privileged node. This node then sends INITIALIZE messages to its immediate neighbors. On receiving the INITIALIZE message, a node sets its HOLDER variable to the node that sent the INITIALIZE message, and send INITIALIZE messages to its own immediate neighbors. Once INITIALIZE message is received, a node can start making privilege requests even if the entire tree is not initialized.

The initialization of the following variables is the same at all nodes:

$$\text{USING} := \text{false},$$

$$\text{ASKED} := \text{false},$$

$$\text{REQUEST\_}Q := \text{empty}.$$

9.12.7 Node failures and recovery

If a node fails, lost information can be reconstructed on restart. Once a node restarts, it enters into a recovery phase and selects a delay period for the recovery phase in order to get back all the lost information. It sends RESTART messages to its immediate neighbors and waits for ADVISE messages. During the recovery phase, the node can still receive REQUEST and PRIVILEGE messages; it acts as any normal node would act in response to those messages except that ASSIGN_PRIVILEGE and MAKE_REQUEST routines are not executed.

The ADVISE message that a recovering node A receives from each immediate neighbor B will contain information on the HOLDER, ASKED, and REQUEST_Q variables of B, from which A can reconstruct its own HOLDER, ASKED, and REQUEST_Q variables.

For example, if $\text{HOLDER}_B = \text{A}$ for all immediate neighbors B of node A, it means node A holds the privilege, and hence $\text{HOLDER}_A = \text{self}$. Similar reasoning can be applied to determine value of ASKED_A and the elements of REQUEST_Q_A. REQUEST_Q_A can be reconstructed but the elements may not be in proper order. To ensure proper order, the ADVISE messages could provide real or logical timestamps for its REQUEST messages. USING_A can be set to false.

The recovering node's REQUEST_Q can have duplicates if it processes REQUEST messages sent currently and the ones it receives in the ADVISE messages. However, this does not affect the working of the algorithm as long as the REQUEST_Q is large enough to accommodate such situations. A node can also possibly fail when recovering from an earlier failure. In such a case, ASSIST messages related to the first recovery phase can be identified by making use of the delay chosen for recovery or unique identifiers, and those messages can be discarded.

9.13 Chapter summary

Mutual exclusion is a fundamental problem in distributed computing systems, where concurrent access to a shared resource or data is serialized. Mutual exclusion in a distributed system requires that only one process be allowed to execute the critical section at any given time. Mutual exclusion algorithms for distributed computing systems have been designed based on three approaches: token-based approach, non-token-based approach, and quorum-based approach. In token-based algorithms, a unique token is shared among the sites and a site is allowed to enter its critical section only if it possesses the token. Depending upon the way the token is managed in the system, there are several token-based algorithms.

In the non-token-based approach, sites exchange two or more rounds of messages to determine which site will enter the critical section next. In the quorum-based approach, each site requests permission 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, at least one site receives both the requests and which is responsible to make sure that only one request executes the critical section at any time.

A large number of mutual exclusion algorithms based on these approaches have been developed. In this chapter, we described a set of representative mutual exclusion algorithms. Early mutual exclusion algorithms were static in the sense they always take the same course of actions to invoke mutual exclusion regardless of the state of the system. These algorithms lack efficiency because these algorithms fail to exploit the changing conditions in the system. Lately, dynamic mutual exclusion algorithms have been developed. Such algorithms exploit dynamic conditions of the system to optimize the performance.

9.14 Exercises

Exercise 9.1 Consider the following simple method to enforce mutual exclusion: all sites are arranged in a logical ring fashion and a unique token circulates around the ring hopping from a site to another site. When a site needs to executes its CS, it waits for the token, grabs the token, executes the CS, and then dispatches the token to the next site on the ring. If a site does not need the token on its arrival, it immediately dispatches the token to the next site (in zero time).

1. What is the reponse time when the load is low?
2. What is the reponse time when the load is heavy?

Assume there are N sites, the message/token delay is T, and the CS execution time is E.

Exercise 9.2 In Lamport's algorithm, condition L1 can hold concurrently at several sites. Why do we need this condition for guaranteeing mutual exclusion?

Exercise 9.3 Show that in Lamport's algorithm if a site S_i is executing the critical section, then S_i's request need not be at the top of the *request_queue* at another site S_j. Is this still true when there are no messages in transit?

Exercise 9.4 What is the purpose of a REPLY message in Lamport's algorithm? Note that it is not necessary that a site must always return a REPLY message in response to a REQUEST message. State the condition under which a site does not have to return REPLY message. Also, give the new message complexity per critical section execution in this case.

Exercise 9.5 Show that in the Ricart–Agrawala algorithm the critical section is accessed in increasing order of timestamp. Does the same hold in Maekawa's algorithm?

Exercise 9.6 Mutual exclusion can be achieved using the following simple method in a distributed system (called the "centralized" mutual exclusion algorithm): to access the shared resource, a site sends the request to the site that contains the resource. This site executes the requests using any classical methods for mutual exclusion (like semaphores).

Discuss what prompted Lamport's mutual exclusion algorithm even though it requires many more messages ($3(N-1)$ as compared to only 3).

Exercise 9.7 Show that in Lamport's algorithm the critical section is accessed in increasing order of timestamp.

Exercise 9.8 Show by examples that the staircase configuration among sites is preserved in Singhal's dynamic mutual exclusion algorithm when two or more sites request the CS concurrently and have executed the CSs.

9.15 Notes on references

Singhal gives a taxonomy on distributed mutual exclusion in [24]. Raynal presents a survey of mutual exclusion algorithms in [20]. A large number of token-based mutual exclusion algorithms have appeared in last several years, e.g., mutual exclusion algorithms by Ahamad and Bernabeu [2], Helary *et al.* [10], Naimi and Trehel [15], Chang *et al.* [6], and Neilsen and Mizuno [16]. In [23], Saxena and Rai present a survey of permission-based distributed mutual exclusion algorithms.

Nishio *et al.* [18] presented a technique for generation of unique token in case of a token loss. A dynamic heuristic-based token mutual exclusion algorithm is given in [26]. Snepscheut [30] extended tree-based algorithms to handle a connected network of any topology (i.e., graphs). Due to network topology, such algorithms are fault-tolerant to site and link failures. Chang *et al.* [7] present a fault-tolerant mutual exclusion algorithm. Goscinski [8] has presented two mutual exclusion algorithms for real-time distributed systems. Coterie-based mutual exclusion algorithms, which are a generalization of Maekawa's $\sqrt{N}$ algorithm, have lately attracted considerable attention. Barbara and Garcia-Molina [9] and Ibaraki and Kameda [11] have discussed theoretical aspects of coteries. Cao and Singhal developed a delay optimal coterie-based mutual exclusion algorithm [5].

Sanders [22] gave the concept of information structures to develop a generalized mutual exclusion algorithm. Other mutual exclusion algorithms can be found in [3, 4, 17, 25].

References

[1] D. Agrawal and A. E. Abbadi, An efficient and fault-tolerant solution for distributed mutual exclusion, *ACM Transactions on Computer Systems*, **9**(1), 1991, 1–20.

[2] J. M. Bernabeu-Auban and M. Ahamad, Applying a path-compression technique to obtain an effective distributed mutual exclusion algorithm, *Proceedings of the 3rd International Workshop on Distributed Algorithms*, September 1989, 33–44.

[3] G. Buckley and A. Silberschatz, A failure tolerant centralized mutual exclusion algorithm, *Proceedings of the 4th International Conference on Distributed Computing Systems*, May 1984, 347–356.

[4] O. S. F. Carvalho and G. Roucairol, On mutual exclusion in real-time distributed computing systems, technical Correspondence, *Communications of the ACM*, **26**(2), 1983, 146–147.

[5] G. Cao and M. Singhal, A delay-optimal quorum-based mutual exclusion algorithm for distributed systems, *IEEE Transactions on Parallel and Distributed Systems*, **12**(12), 2001, 1256–1268.

[6] Y. Chang, M. Singhal, and M. Liu, A dynamic token-based distributed mutual exclusion algorithm, *Proceedings of the 10th IEEE International Phoenix Conference on Computer and Communications*, March 1991, 240–246.

[7] Y. Chang, M. Singhal, and M. Liu, A fault-tolerant mutual exclusion algorithm for distributed systems, *Proceedings of the 9th Symposium on Reliable Distributed Software and Systems*, October 1990, 146–154.

[8] A. Goscinski, Two algorithms for mutual exclusion in real-time distributed computing systems, *Journal of Parallel and Distributed Computing*, **9**(1), 1990, 77–82.

[9] H. Garcia-Molina and D. Barbara, How to assign votes in a distributed system, *Journal of the ACM*, 1985.

[10] M. Helary, N. Plouzeau, and M. Raynal A distributed algorithm for mutual exclusion in an arbitrary network, *Computing Journal*, **31**(4), 1988, 289–295.

[11] T. Ibaraki and T. Kameda, *Theory of Coteries*, Technical Report, CSS/LCCR TR90-09, University of Kyoto, Kyoto, Japan, 1990.

[12] L. Lamport Time, clocks and ordering of events in distributed systems, *Communications of the ACM*, **21**(7), 1978, 558–565.

[13] S. Lodha and A. Kshemkalyani, A fair distributed mutual exclusion algorithm, *IEEE Transactions on Parallel and Distributed Systems*, **11**(6), 2000, 537–549.

[14] M. Maekawa, A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems, *ACM Transactions on Computer Systems*, **3**(2), 1995, 145–159.

[15] M. Naimi and M. Trehel, An improvement of the $\log N$ distributed algorithm for mutual exclusion, *Proceedings of the 7th International Conference on Distributed Computing Systems*, September 23–25, 1987, 371–377.

[16] M. L. Neilsen and M. Mizuno, A DAG-based algorithm for distributed mutual exclusion, *Proceedings of the 11th International Conference on Distributed Computing Systems*, May 21–23, 1991, 354–360.

[17] M. Nesterenko and M. Mizuno, A quorum-based self-stabilizing distributed mutual exclusion algorithm, *Journal of Parallel and Distributed Computing*, **62**(2), 2002, 284–305.

[18] S. Nishio, K. F. Li, and E. G. Manning, A resilient mutual exclusion algorithm for computer networks, *IEEE Transactions on Parallel and Distributed Systems*, **1**(3), 1990, 344–356.

[19] K. Raymond, Tree-based algorithm for distributed mutual exclusion, *ACM Transactions on Computer Systems*, **7**, 1989, 61–77.

[20] M. Raynal, A simple taxonomy of distributed mutual exclusion algorithms, *Operating Systems Review*, **25**(2), 1991, 47–50.

[21] G. Ricart and A. K. Agrawala, An optimal algorithm for mutual exclusion in computer networks, *Communications of the ACM*, **24**(1), 1981, 9–17.

[22] B. Sanders, The information structure of distributed mutual exclusion algorithms, *ACM Transactions on Computer Systems*, **5**(3), 1987, 284–299.

[23] P. C. Saxena and J. Rai, A survey of permission-based distributed mutual exclusion algorithms, *Computer Standards and Interfaces*, **25**(2), 2003, 159–181.

[24] M. Singhal, A taxonomy of distributed mutual exclusion, *Journal of Parallel and Distributed Computing*, **18**(1), 1993, 94–101.

[25] M. Singhal, "A class of deadlock-free Maekawa type mutual exclusion algorithms for distributed systems", *Distributed Computing*, **4**(3), 1991, 131–138.

[26] M. Singhal, A heuristically-aided algorithm for mutual exclusion in distributed systems, *IEEE Transactions on Computers*, **38**(5), 1989, 651–662.

[27] M. Singhal, A dynamic information structure mutual exclusion algorithm for distributed systems, *Proceedings of the 9th International Conference on Distributed Computing Systems*, June 5–9, 1989, Newport Beach, CA, 70–78.

[28] M. Singhal, A dynamic information-structure mutual exclusion algorithm for distributed systems, *IEEE Transactions on Parallel and Distributed Systems*, **3**(1), 1992, 121–125.

[29] I. Suzuki and T. Kasami, A distributed mutual exclusion algorithm, *ACM Transactions on Computer Systems*, **3**(4), 1985, 344–349.

[30] J. L. A. Vas de Snepscheut, Fair mutual exclusion on a graph of processes, *Distributed Computing*, **2**, 1987, 113–115.