6.1 INTRODUCTION

In the problem of mutual exclusion, concurrent access to a shared resource by several uncoordinated user-requests is serialized to secure the integrity of the shared resource. It requires that the actions performed by a user on a shared resource must be atomic. That is, if several users concurrently access a shared resource then the actions performed by a user, as far as the other users are concerned, must be instantaneous and indivisible such that the net effect on the shared resource is the same as if user actions were executed serially, as opposed to in an interleaved manner.

The problem of mutual exclusion frequently arises in distributed systems whenever concurrent access to shared resources by several sites is involved. For correctness, it is necessary that the shared resource be accessed by a single site (or process) at a time. A typical example is directory management, where an update to a directory must be done atomically because if updates and reads to a directory proceed concurrently, reads may obtain inconsistent information. If an entry contains several fields, a read operation may read some fields before the update and some after the update. Mutual exclusion is a fundamental issue in the design of distributed systems and an efficient and robust technique for mutual exclusion is essential to the viable design of distributed systems.
Mutual exclusion in single-computer systems vs. distributed systems

The problem of mutual exclusion in a single-computer system, where shared memory exists, was studied in Chap. 2. In single-computer systems, the status of a shared resource and the status of users is readily available in the shared memory, and solutions to the mutual exclusion problem can be easily implemented using shared variables (e.g., semaphores). However, in distributed systems, the shared resources and the users may be distributed and shared memory does not exist. Consequently, approaches based on shared variables are not applicable to distributed systems and approaches based on message passing must be used.

The problem of mutual exclusion becomes much more complex in distributed systems (as compared to single-computer systems) because of the lack of both shared memory and a common physical clock and because of unpredictable message delays. Owing to these factors, it is virtually impossible for a site in a distributed system to have current and complete knowledge of the state of the system.

6.2 THE CLASSIFICATION OF MUTUAL EXCLUSION ALGORITHMS

Over the last decade, the problem of mutual exclusion has received considerable attention and several algorithms to achieve mutual exclusion in distributed systems have been proposed. They tend to differ in their communication topology (e.g., tree, ring, and any arbitrary graph) and in the amount of information maintained by each site about other sites. These algorithms can be grouped into two classes. The algorithms in the first class are nontoken-based, e.g., [4, 9, 10, 16, 19]. These algorithms require two or more successive rounds of message exchanges among the sites. These algorithms are assertion based because a site can enter its 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.

The algorithms in the second class are token-based, e.g., [11, 14, 20, 21, 22]. In these algorithms, 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. These algorithms essentially differ in the way a site carries out the search for the token.

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.

6.3 PRELIMINARIES

We now describe the underlying system model and requirements that mutual exclusion algorithms should meet. We also introduce terminology that is used in describing the performance of mutual exclusion algorithms.

SYSTEM MODEL

A system consists of a limited number of sites that are connected together by a network. Each site queues up the following threads: a separate process for the CS (i.e., \( \text{idle} \)), a process to handle CS requests, and a process to handle messages. Each site maintains a local state that is used for the mutual exclusion algorithm. The systems are assumed to be fault-tolerant, which means that the system can continue to process messages even if a process fails. The failure of a site does not disrupt service to other sites. There is no provision for recovery from disruptions.

6.3.1 Requirements

The primary requirement of a mutual exclusion algorithm is that it should prevent two or more processes from entering the CS simultaneously. To meet this requirement, the following characteristics are desirable:

**Freedom**: To execute a CS, a process should get an opportunity to enter the CS.

**Freedom**: If a process has made a request to execute CS with an assertion, other processes should get an opportunity to execute the CS.

**Fairness**: If more than one process makes a request, the process that was last to make a request should be the first to execute the CS.

**Fault Tolerance**: If a process is not able to execute the CS due to a failure, it can resume execution without being interrupted.

**6.3.2 How to Compare Mutual Exclusion Algorithms**

The performance of mutual exclusion algorithms is measured by four metrics: **synchronization overhead**, **total number of messages**, **response time**, and **average critical section execution time**. Synchronization overhead is the ratio of the message exchange to the total execution time of the CS. Total number of messages is the number of messages that are sent and received by a site. Response time is the average time taken by a process to execute the CS. Average critical section execution time is the average time taken by a process to execute the CS. These metrics can be used to evaluate the performance of mutual exclusion algorithms.
DISTRIBUTED MUTUAL EXCLUSION 123

SYSTEM MODEL. At any instant, a site may have several requests for CS. A site queues up these requests and serves them one at a time. A site can be in one of the following three states: requesting CS, executing CS, or neither requesting nor executing CS (i.e., idle). In the requesting CS state, the site is blocked and cannot make further requests for CS. In the idle state, the site is executing outside its 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 a state is referred to as an idle token state.

6.3.1 Requirements of Mutual Exclusion Algorithms

The primary objective of a mutual exclusion algorithm is to maintain mutual exclusion; that is, to guarantee that only one request accesses the CS at a time. In addition, the following characteristics are considered important in a mutual exclusion algorithm:

Freedom from Deadlocks. Two or more sites should not endlessly wait for messages that will never arrive.

Freedom from Starvation. A site should not be forced to wait indefinitely to execute CS while other sites are repeatedly executing CS. That is, every requesting site should get an opportunity to execute CS in a finite time.

Fairness. Fairness dictates that requests must be executed in the order they are made (or the order in which they arrive in the system). Since a physical global clock does not exist, time is determined by logical clocks. Note that fairness implies freedom from starvation, but not vice-versa.

Fault Tolerance. A mutual exclusion algorithm is fault-tolerant if in the wake of a failure, it can reorganize itself so that it continues to function without any (prolonged) disruptions.

6.3.2 How to Measure the Performance

The performance of mutual exclusion algorithms is generally measured by the following four metrics: First, the number of messages necessary per CS invocation. Second, the synchronization delay, which is the time required after a site leaves the CS and before the next site enters the CS (see Fig. 6.1). Note that normally one or more sequential message exchanges are required after a site exits the CS and before the next site enters the CS. Third, the response time, which is the time interval a request waits for its CS execution to be over after its request messages have been sent out (see Fig. 6.2). Thus, response time does not include the time a request waits at a site before its request messages have been sent out. Fourth, the system throughput, which 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:

\[
\text{system throughput} = \frac{1}{sd + E}
\]
LOW AND HIGH LOAD PERFORMANCE. Performance of a mutual exclusion algorithm depends upon the loading conditions of the system and is often studied under two special loading conditions, viz., low load and high load. Under low load conditions, there is seldom more than one request for mutual exclusion simultaneously in the system. Under high load conditions, there is always a pending request for mutual exclusion at a site. Thus, after having executed a request, a site immediately initiates activities to let the next site execute its CS. A site is seldom in an idle state under high load conditions. For many mutual exclusion algorithms, the performance metrics can be easily determined under low and high loads through simple 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 algorithms the best value of the response time is a round-trip message delay plus CS execution time, \(2T + E\) (where \(T\) is the average message delay and \(E\) is the average critical section execution time).

Often for mutual exclusion algorithms, the best and worst cases coincide with low and high loads, respectively. For example, the best and worst values of the response time are achieved when the load is, respectively, low and high. The best and the worse message traffic is generated in Maekawa's algorithm [10] at low and high load conditions, respectively. When the value of a performance metric fluctuates statistically, we generally talk about the average value of that metric.

6.4 A SIMPLE EXCLUSION

In a simple exclusion scheme, a site assigned the token sends a REQUEST message for the CS to the next site in the exclusion ring.

This means that in case of failure, the token does not work. Also, a site should become a bad site if a site sent a request message to it, the site should grant permission to it to enter the CS. Furthermore, the system should be synchronized to \(1/(T + E)\) where \(T\) is the average round-trip delay time and \(E\) is the average critical section execution time.

6.5 NON-TOKEN EXCLUSION

In non-token exclusion algorithms, sites to arbitrate for the CS are identified with the tokens of all the sites. Next, we discuss the priority scheme [9].

Non-token exclusion algorithms for the CS are based on the priority schemes of these algorithms. The priority scheme [9].

6.6 LAMPORT'S ALGORITHM

Lamport was the first to present his clock scheme.

\[ \{S_1, S_2, ..., S_n\} \]

The exclusion request is delivered in the order of the tokens.

The Algorithm

1. When a site requests the CS, it sends a REQUEST message to the next site in the exclusion ring.
6.4 A SIMPLE SOLUTION TO DISTRIBUTED MUTUAL EXCLUSION

In a simple solution to distributed mutual exclusion, a site, called the control site, is assigned the task of granting permission for the CS execution. To request the CS, a site sends a REQUEST message to the control site. The control site queues up the requests for the CS and grants them permission, one by one. This method to achieve mutual exclusion in distributed systems requires only three messages per CS execution.

This naive, centralized solution has several drawbacks. First, there is a single point of failure, the control site. Second, the control site is likely to be swamped with extra work. Also, the communication links near the control site are likely to be congested and become a bottleneck. Third, the synchronization delay of this algorithm is 21' because a site should first release permission to the control site and then the control site should grant permission to the next site to execute the CS. This has serious implications for the system throughput, which is equal to 1/(21' + E) in this algorithm. Note that if the synchronization delay is reduced to 1', the system throughput is almost doubled to 1/(1' + E). We later discuss several mutual exclusion algorithms that reduce the synchronization delay to 1' at the cost of higher message traffic.

6.5 NON-TOKEN-BASED ALGORITHMS

In non-token-based mutual exclusion algorithms, a site communicates with a set of other sites to arbitrate who should execute the CS next. For a site Si, request set Ri contains ids of all those sites from which site Si must acquire permission before entering the CS. Next, we discuss some non-token-based mutual exclusion algorithms which are good representatives of this class.

Non-token-based mutual exclusion algorithms use timestamps to order requests for the CS and to resolve conflicts between simultaneous requests for the CS. In all these algorithms, logical clocks are maintained and updated according to Lamport's scheme [9]. Each request for the CS gets a timestamp, and smaller timestamp requests have priority over larger timestamp requests.

6.6 LAMPORT'S ALGORITHM

Lamport was the first to give a distributed mutual exclusion algorithm as an illustration of his clock synchronization scheme [9]. In Lamport's algorithm, ∀i : 1 ≤ i ≤ N :: Ri = {S1, S2, ..., SN}. Every site Si keeps a queue, request_queuei, which contains mutual exclusion requests ordered by their timestamps. This algorithm requires messages to be delivered in the FIFO order between every pair of sites.

The Algorithm

Requesting the critical section.

1. When a site Si wants to enter the CS, it sends a REQUEST(ts_i, i) message to all the sites in its request set Ri and places the request on request_queuei. ((ts_i, i) is the timestamp of the request.)
2. When a site $S_j$ receives the REQUEST($t_{s_i}, i$) message from site $S_i$, it returns a timestamped REPLY message to $S_i$ and places site $S_i$'s request on request_queue.

**Executing the critical section.** Site $S_i$ enters the CS when the two following conditions hold:

[L1:] $S_i$ has received a message with timestamp larger than ($t_{s_i}, i$) from all other sites.

[L2:] $S_i$'s request is at the top of request_queue.

**Releasing the critical section.**

3. Site $S_i$, upon exiting the CS, removes its request from the top of its request queue and sends a timestamped RELEASE message to all the sites in its request set.

4. When a site $S_j$ receives a RELEASE message from site $S_i$, it removes $S_i$'s request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS. The algorithm executes CS requests in the increasing order of timestamps.

**Correctness**

**Theorem 6.1.** Lamport's algorithm achieves mutual exclusion.

**Proof:** The 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 a loss of generality, assume that $S_i$'s request has a smaller timestamp than the request of $S_j$. Due to condition L1 and the FIFO property of the communication channels, it is clear that at instant $t$, the request of $S_i$ must be present in request_queue, 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—a contradiction! Hence, Lamport's algorithm achieves mutual exclusion. □

**Example 6.1.** In Fig. 6.3 through Fig. 6.6, we illustrate the operation of Lamport's algorithm. In Fig. 6.3, 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 (2, 1) and (1, 2), respectively. In Fig. 6.4, $S_2$ has received REPLY messages from all the other sites and its request is at the top of its request_queue. Consequently, it enters the CS. In Fig. 6.5, $S_2$ exits and sends RELEASE messages to all other sites. In Fig. 6.6, site $S_1$ has received REPLY messages from all other sites and its request is at the top of its request_queue. Consequently, it enters the CS next.
A request may come from any site, it returns a request.queue.j.

Sj's request set.

The request may come
from any site, it returns
request.queue.

Sj's request
set.

Sj's request
set.

FIGURE 6.3
Sites S1 and S2 are making requests for the CS.

PERFORMANCE. Lamport's algorithm requires \(3(N-1)\) messages per CS invocation: \((N-1)\) REQUEST, \((N-1)\) REPLY, and \((N-1)\) RELEASE messages. Synchronization delay in the algorithm is \(T\).

AN OPTIMIZATION. Lamport's algorithm can be optimized to require between \(3(N-1)\) and \(2(N-1)\) messages per CS execution by suppressing REPLY messages.
in certain situations. For example, suppose site $S_j$ receives a REQUEST message from site $S_i$ after it has sent its own REQUEST message with timestamp higher than the timestamp of site $S_i$'s request. In this case, 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 that is still pending (because the communication medium preserves message ordering).

6.7 THE RICART-AGRAWALA ALGORITHM

The Ricart-Agrawala algorithm [16] is an optimization of Lamport's algorithm that dispenses with RELEASE messages by cleverly merging them with REPLY messages. In this algorithm also, $\forall i: 1 \leq i \leq N :: R_i = \{S_1, S_2, ..., S_N\}$.

The Algorithm

Requesting the critical section.

1. When a site $S_i$ wants to enter the CS, it sends a timestamped REQUEST message to all the sites in its request set.
2. 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 site $S_j$ is requesting and $S_i$'s request's timestamp is smaller than site $S_j$'s own request's timestamp. The request is deferred otherwise.

Executing the critical section

3. Site $S_i$ enters the CS after it has received REPLY messages from all the sites in its request set.

Releasing the critical section

4. When site $S_i$ exits the CS, it sends REPLY messages to all the deferred requests.
A site’s REPLY messages are blocked only by sites that are requesting the CS with higher priority (i.e., a smaller timestamp). Thus, when a site sends out REPLY messages to all the deferred requests, the site with the next highest priority request receives the last needed REPLY message and enters the CS. The execution of CS requests in this algorithm is always in the order of their timestamps.

**CORRECTNESS**

**Theorem 6.2.** The Ricart-Agrawala algorithm achieves mutual exclusion.

**Proof:** The proof is by contradiction. Suppose two sites $S_i$ and $S_j$ are executing the CS concurrently and $S_i$’s request has a higher priority (i.e., a smaller timestamp) than the request of $S_j$. Clearly, $S_i$ received $S_j$’s request after it had made its own request. (Otherwise, $S_i$’s request would 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 6.2.** Figures 6.7 through 6.10 illustrate the operation of the Ricart-Agrawala algorithm. In Fig. 6.7, sites $S_1$ and $S_2$ are making requests for the CS, sending out REQUEST messages to other sites. The timestamps of the requests are $(2, 1)$ and $(1, 2)$, respectively. In Fig. 6.8, $S_2$ has received REPLY messages from all other sites and consequently, it enters the CS. In Fig. 6.9, $S_2$ exits the CS and sends a REPLY message to site $S_1$. In Fig. 6.10, site $S_3$ has received REPLY messages from all other sites and enters the CS next.

**PERFORMANCE.** The Ricart-Agrawala algorithm requires $2(N - 1)$ messages per CS execution: $(N - 1)$ REQUEST and $(N - 1)$ REPLY messages. Synchronization delay in the algorithm is $T$.

**AN OPTIMIZATION.** Roucairol and Carvalho [4] proposed an improvement to the Ricart-Agrawala algorithm by observing that once a site $S_i$ has received a REPLY message from a site $S_j$, the authorization implicit in this message remains valid until $S_i$
6.8 MAEKAWA'S ALGORITHM

Maekawa's algorithm consists of the following steps:

- **Requesting the critical section**: A site $S_i$ requests the critical section by sending a REQUEST message to all other sites.

- **Granting permission**: Each site $S_j$ grants permission by sending a REPLY message to the requesting site $S_i$.

- **Entering the critical section**: After receiving a REPLY message from all other sites, site $S_i$ enters the critical section.

- **Exiting the critical section**: Site $S_i$ exits the critical section after the request is granted by all other sites.

- **Forcing non-conflicting sites** to exit**: Site $S_i$ forces all other sites that do not conflict with the critical section to exit.

The Algorithm

Maekawa's algorithm consists of the following steps:

1. **Requesting the critical section**: A site $S_i$ requests the critical section by sending a REQUEST message to all other sites.
2. **Granting permission**: Each site $S_j$ grants permission by sending a REPLY message to the requesting site $S_i$.
3. **Entering the critical section**: After receiving a REPLY message from all other sites, site $S_i$ enters the critical section.
4. **Exiting the critical section**: Site $S_i$ exits the critical section after the request is granted by all other sites.
5. **Forcing non-conflicting sites** to exit**: Site $S_i$ forces all other sites that do not conflict with the critical section to exit.

The Algorithm

Maekawa's algorithm consists of the following steps:

1. **Requesting the critical section**: A site $S_i$ requests the critical section by sending a REQUEST message to all other sites.
2. **Granting permission**: Each site $S_j$ grants permission by sending a REPLY message to the requesting site $S_i$.
3. **Entering the critical section**: After receiving a REPLY message from all other sites, site $S_i$ enters the critical section.
4. **Exiting the critical section**: Site $S_i$ exits the critical section after the request is granted by all other sites.
5. **Forcing non-conflicting sites** to exit**: Site $S_i$ forces all other sites that do not conflict with the critical section to exit.

The Algorithm

Maekawa's algorithm consists of the following steps:

1. **Requesting the critical section**: A site $S_i$ requests the critical section by sending a REQUEST message to all other sites.
2. **Granting permission**: Each site $S_j$ grants permission by sending a REPLY message to the requesting site $S_i$.
3. **Entering the critical section**: After receiving a REPLY message from all other sites, site $S_i$ enters the critical section.
4. **Exiting the critical section**: Site $S_i$ exits the critical section after the request is granted by all other sites.
5. **Forcing non-conflicting sites** to exit**: Site $S_i$ forces all other sites that do not conflict with the critical section to exit.

The Algorithm

Maekawa's algorithm consists of the following steps:

1. **Requesting the critical section**: A site $S_i$ requests the critical section by sending a REQUEST message to all other sites.
2. **Granting permission**: Each site $S_j$ grants permission by sending a REPLY message to the requesting site $S_i$.
3. **Entering the critical section**: After receiving a REPLY message from all other sites, site $S_i$ enters the critical section.
4. **Exiting the critical section**: Site $S_i$ exits the critical section after the request is granted by all other sites.
5. **Forcing non-conflicting sites** to exit**: Site $S_i$ forces all other sites that do not conflict with the critical section to exit.

The Algorithm

Maekawa's algorithm consists of the following steps:

1. **Requesting the critical section**: A site $S_i$ requests the critical section by sending a REQUEST message to all other sites.
2. **Granting permission**: Each site $S_j$ grants permission by sending a REPLY message to the requesting site $S_i$.
3. **Entering the critical section**: After receiving a REPLY message from all other sites, site $S_i$ enters the critical section.
4. **Exiting the critical section**: Site $S_i$ exits the critical section after the request is granted by all other sites.
5. **Forcing non-conflicting sites** to exit**: Site $S_i$ forces all other sites that do not conflict with the critical section to exit.
6.8 MAEKAWA'S ALGORITHM

Maekawa's algorithm [10] is a departure from the general trend in the following two ways: First, a site does not request permission from every other site, but only from a subset of the sites. This is a radically different approach as compared to the Lamport and the Ricart-Agrawala algorithms, where all sites participate in the conflict resolution of all other sites. In Maekawa's algorithm, the request set of each site is chosen such that

\[ \forall i \forall j : 1 \leq i, j \leq N :: R_i \cap R_j \neq \emptyset \]

Consequently, every pair of sites has a site that mediates conflicts between that pair. Second, in Maekawa's algorithm a site can send out only one REPLY message at a time. A site can only 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.

THE CONSTRUCTION OF REQUEST SETS. The request sets for sites in Maekawa's algorithm are constructed to satisfy the following conditions:

- **M1:** \( (\forall i \forall j : i \neq j, 1 \leq i, j \leq N :: R_i \cap R_j \neq \emptyset) \)
- **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 \( R_i \)'s, \( 1 \leq i, j \leq N \). Maekawa established the following relation between \( N \) and \( K \): \( N = K(K-1) + 1 \). 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 that 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 the 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 request sets of all the sites must be equal, implying that all sites should have to do an equal amount of work to invoke mutual exclusion. Condition M4 enforces that exactly the same number of sites should request permission from any site, implying that all sites have equal responsibility in granting permission to other sites.

The Algorithm

Maekawa's algorithm works in the following manner:

**Requesting the critical section.**

1. A site \( S_i \) requests access to the CS by sending REQUEST\((i)\) messages to all the sites in its request set \( R_i \).
2. 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 from the time it received the last RELEASE message. Otherwise, it queues up the REQUEST for later consideration.

Executing the critical section.

3. Site $S_i$ accesses the CS only after receiving REPLY messages from all the sites in $R_i$.

Releasing the critical section.

4. After the execution of the CS is over, site $S_i$ sends RELEASE($i$) message to all the sites in $R_i$.

5. 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 the site has not sent out any REPLY message.

CORRECTNESS

Theorem 6.3. Maekawa’s algorithm achieves mutual exclusion.

Proof: The proof is by contradiction. Suppose two sites $S_i$ and $S_j$ are concurrently executing the CS. 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, the execution of a 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 $2T$. As discussed next, Maekawa’s algorithm is deadlock-prone. Measures to handle deadlocks require additional messages.

THE PROBLEM OF DEADLOCKS. Maekawa’s algorithm is prone to deadlocks because a site is exclusively locked by other sites and requests are not prioritized by their timestamps [10, 17]. Without the 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, it is possible that due to arbitrary message delays, $S_{ij}$ 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 $S_{ki}$) resulting in a deadlock involving the 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 on the lock. The site in turn checks if the needed sites have already sent out REQUEST messages to the needed sites. If so, the site proceeds with its REPLY messages; otherwise, it has already locked the lock and may send out any messages:

FAILED. It grants $S_j$’s request and sends a REPLY message to it. If it receives a REPLY from $S_j$, it sends a priority request.

INQUIRY. A site finds out from $S_j$ whether the priority request has been granted permission to $S_j$.

YIELD. A site $S_i$ finds out from $S_j$ permission to $S_j$, and sends $S_j$ a REPLY message to $S_i$.

Details of messages:

- When a REQUEST message is granted permission to a request has been granted permission to $S_j$’s
- In response to the request, a site sends a message to $S_j$ to resolve the request set of sites that waits REPLY from $S_j$.
- In response to the request, a site sends a request set of sites that waits REPLY from $S_j$.
- In response to the request, a site sends a request set of sites that waits REPLY from $S_j$.

Thus, Maekawa’s algorithm may exchange these messages for resolving of messages required to resolve deadlocks.

6.9 A GENERATION OF THEOREMS

Sanders gave a generalization of the Maekawa’s algorithm for unifying different operating systems [17] and derived some special cases of the problem for unifying different operating systems.

6.9.1 Information structure

The information structure needed at a site to resolve the problem is the information structure needed for unifying different operating systems.
other request waiting for the same lock (unless the former has succeeded in locking all
the needed sites) [10, 17]. A site suspects a deadlock (and initiates message exchange
to resolve it) whenever a higher priority request finds that a lower priority request
has already locked the site. 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 the deadlock handling steps 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 or 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 it has been
  released by $S_k$, places the request of $S_k$ at the 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 $5\sqrt{N}$.

6.9 A GENERALIZED NON-TOKEN-BASED ALGORITHM
Sanders gave a generalized non-token-based mutual exclusion algorithm for distributed
systems [17] and all the existing non-token-based mutual exclusion algorithms are
special cases of this algorithm. The concept of information structure forms the basis
for unifying different non-token-based mutual exclusion algorithms.

6.9.1 Information Structures
The information structure of a mutual exclusion algorithm defines the data structure
needed at a site to record the status of other sites. The information kept in the infor­
mation structure is used by a site in making decisions (i.e., from which sites to request
permission) when invoking mutual exclusion.
The information structures at a site $S_i$ consists of the following three sets: a request set $R_i$, an inform set $I_i$, and a status set $St_i$. These sets consist of the ids of the sites of the system. A site must acquire permission from all the sites in its request set before entering CS. Every site must inform all the sites in its inform set of its status change due to the wait to enter the CS and due to the exit from the CS. The status set $St_i$ contains the ids of sites for which $S_i$ maintains status information. Note that the inform set and the status set are dependent on each other. The contents of one is decided by the contents of the other. If $S_i \in I_j \Rightarrow S_j \in St_i$.

A site also maintains a variable CSSTAT, which indicates the site’s knowledge of the status of the CS. Every site maintains a queue which contains REQUEST messages, in the order of their timestamps, for which no GRANT message has been sent.

**CORRECTNESS CONDITION.** To guarantee mutual exclusion, the information structure of sites in the generalized algorithm must satisfy the conditions given by the following theorem [17]:

**Theorem 6.4.** If $\forall i: 1 \leq i \leq N:: S_i \in I_i$, then the following two conditions are necessary and sufficient to guarantee mutual exclusion:

\[
\begin{align*}
\text{G1: } & \forall i: 1 \leq i \leq N :: I_i \subseteq R_i \\
\text{G2: } & \forall \forall j: 1 \leq i, j \leq N :: (I_i \cap I_j \neq \emptyset) \lor (S_i \in R_j \land S_j \in R_i)
\end{align*}
\]

The correctness condition G2 states that for every pair of sites, either they request permission from each other or they request permission from a common site (which maintains the status information of both).

### 6.9.2 The Generalized Algorithm

In the generalized algorithm, each request for a CS is assigned a timestamp that is maintained according to Lamport’s scheme [9]. Timestamps are used to prioritize requests in case of conflicts.

**Requesting the critical section.**

1. To execute CS, a site sends timestamped REQUEST messages to all the sites in its request set.

2. On the receipt of a REQUEST message, a site $S_i$ takes the following actions:
   - It places the request in its queue (which is ordered by timestamps).
   - If CSSTAT indicates that the CS is free, then it sends a GRANT message to the site at the top of the queue and removes its entry from the queue. If the recipient of the GRANT message is in $St_i$, then CSSTAT is set to indicate that the site is in CS.

**Executing the critical section.**

3. A site executes CS only if its queue contains REQUEST messages from all the sites in its request set. On receiving a GRANT message:
   - CSSTAT is set.
   - If its queue is empty then its message is removed from the queue.
   - The previous site, or its queue, is informed of the exit from CS.

**Releasing the critical section.**

4. On exiting the CS, a site sends a REQUEST message to all the sites in its request set. On receiving a REQUEST message:
   - CSSTAT is cleared.
   - If its queue contains REQUEST messages, then its queue length is decremented.
   - The previous site, or its queue, is informed of the exit from CS.

The proof of the correctness of the generalized algorithm is left to the readers.
Executing the critical section.

3. A site executes the CS only after it has received a GRANT message from all the sites in its request set.

Releasing the critical section.

4. On exiting the CS, the site sends a RELEASE message to every site in its inform set. On receiving a RELEASE message, a site \( S_i \) takes the following actions:
   - CSSTAT is set to free.
   - If its queue is nonempty, then it sends a GRANT message to the site at the top of the queue and removes its entry from the queue. If the recipient of the GRANT message is in \( S_i \), then CSSTAT is set to indicate that the site is in the CS.
   - The previous action is repeated until CSSTAT indicates that a site is in the CS or its queue becomes empty.

The proof of correctness is quite involved and is therefore omitted. Interested readers are encouraged to refer to the original paper [17].

DISCUSSION OF THE GENERALIZED ALGORITHM. The generalized algorithm combines the strategies of the Ricart-Agrawala algorithm [16] and Maekawa’s [10] algorithm. In both these algorithms, a site invoking mutual exclusion acquires permission from a set of sites. However, the semantics of permission are very different in both the algorithms. In the Ricart-Agrawala algorithm, a site can grant permission to many sites simultaneously. A site grants permission to a requesting site immediately if it is not requesting the CS or its own request has lower priority. Otherwise, it defers granting permission until its execution of the CS is over. The Semantics of granting permission in this algorithm is essentially, “As far as I am concerned, it is OK for you to enter the CS.” A site handles a REQUEST message to take care of its mutual exclusion with respect to all other sites.

In Maekawa’s algorithm, a site can grant permission only to one site at a time. A site grants permission to a site only if it has not currently granted permission to another site. Otherwise, it delays granting permission until the currently granted permission has been released. Thus, acquiring permission is like locking the site in the exclusive mode. The semantics of granting permission in this algorithm is “As far as all the sites in my status set are concerned, it is OK for you to enter the CS.” A site \( S_i \) handles a REQUEST message so that the requesting site can have mutual exclusion with respect to sites in \( S_i \)’s status set. By granting permission to a site, the site guarantees that no other sites in its status set can execute the CS concurrently.

In the generalized algorithm, a site \( S_i \) acquires permission of the Maekawa type from all the sites in its inform set \( I_i \), and acquires permission of the Ricart-Agrawala type from all the sites in set \( R_i - I_i \). In response to a REQUEST message, a site \( S_j \) sends Maekawa type permission to sites in its status set and sends Ricart-Agrawala type permission to all the other sites. After a site has granted Maekawa type permission, it cannot grant permission to any other site unless it has been released. However, in the generalized algorithm a site can concurrently grant many Ricart-Agrawala type permissions preceding a Maekawa type permission.
If the first predicate of correctness condition $G_2$ is false for all $S_i$ and $S_j$, then the resulting algorithm of the Ricart-Agrawala type. If the second predicate of condition $G_2$ is false for all $S_i$ and $S_j$, then the resulting algorithm is of the Maekawa type. If the first predicate is true for some sites and the second predicate is true for other sites, then the resulting algorithm is a generalized mutual exclusion algorithm.

**Example 6.3.** Figure 6.11 illustrates three mutual exclusion algorithms in terms of their information structures [17]. A solid arrow from $S_i$ to $S_j$ indicates that $S_j \in I_i$ and $S_i \in R_j$. A dashed arrow from $S_i$ to $S_j$ indicates that $S_j \in R_i$ and $S_i \notin I_j$. Figure 6.11(a) shows the information structure of a mutual exclusion algorithm where a single site, $S_1$, controls entry into the CS [3]. Figure 6.11(b) shows the information structure of a mutual exclusion algorithm where every site requests permission of every other site to enter the CS and a site maintains information about its own status. An example of such an algorithm is the Ricart-Agrawala algorithm [16]. Figure 6.11(c) shows the information structure of Maekawa’s mutual exclusion algorithm with 4 sites. Note that in this case, the request set and the inform set of every site is identical and $\forall i \forall j : i \neq j, 1 \leq i, j \leq 4 : R_i \cap R_j \neq \phi$.  

![Diagram](image_url)

**FIGURE 6.11** Examples of information structures.

---

### 6.9.3 Static

Non-token-based algorithms use dynamic information structures in which the contents of request sets and inform sets of sites execute concurrently. If the contents of the request sets and inform sets of the Ricart-Agrawala algorithm is found in [4] and the Maekawa algorithm is found in [5], then the information structure in Figure 6.11 is the reason these algorithms are successful for mutual exclusion.

### 6.10 Token-based

In token-based algorithms, a site controls entry into the CS. Before we explain how sites execute concurrently, we must distinguish between some representative algorithms.

Before we proceed, we determine that a site has the token before it enters the CS. Every request set must be able to find the token in order: First, every entry request is processed. Then, before entering the CS, it makes a request for the token. This way, we can distinguish between some representative algorithms to ensure that the token guarantees mutual exclusion when sites execute concurrently. Rather, the token is prominent.

### 6.11 Suzuki

In the Suzuki-Kajikawa algorithm, every site that possesses the token sends a QUEST message. If a site does not have the token, it requests the token. If a site possesses the token only and does not have the token, it repeatedly until it is granted.

The main messages in the following token-based algorithms are the QUEST message, requesting its $n^{th}$ token.
6.9.3 Static vs. Dynamic Information Structures

Non-token-based mutual exclusion algorithms can be classified as either static or dynamic information structure algorithms. In static information structure algorithms, the contents of request sets, inform sets, and status sets remain fixed and do not change as sites execute CS. Examples of such algorithms are Lamport’s [9], Maekawa’s [10], and Ricart-Agrawala’s [16] algorithms. In dynamic information structure algorithms, the contents of these sets change as the sites execute CS. Examples of such algorithms are found in [4] and [19]. The design of dynamic information structure mutual exclusion algorithms is much more complex because it requires rules for updating the information structure such that the conditions for mutual exclusion are always satisfied. This is the reason that most mutual exclusion algorithms for distributed systems have static information structures.

6.10 Token-Based Algorithms

In token-based algorithms, a unique token is shared among all sites. A site is allowed to enter its CS if it possesses the token. Depending upon the way a site carries out its search for the token, there are numerous token-based algorithms. Next, we discuss some representative 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, a correctness proof of token-based algorithms to ensure that mutual exclusion is enforced is trivial because an algorithm guarantees mutual exclusion so long as a site holds the token during the execution of the CS. Rather, the issues of freedom from starvation and freedom from deadlock are prominent.

6.11 Suzuki-Kasami’s Broadcast Algorithm

In the Suzuki-Kasami’s algorithm [21], if a site attempting to enter the CS does not have the token, it broadcasts a REQUEST message for the token to all the other sites. A site that possesses the token sends it to the requesting site upon receiving its REQUEST message. If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has exited the CS. A site holding the token can enter its CS repeatedly until it sends the token to some other site.

The main design issues in this algorithm are: (1) distinguishing outdated REQUEST messages from current REQUEST messages and (2) determining which site has an outstanding request for the CS.

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 $n^{th}$ CS execution. A site $S_i$ keeps an array of integers $RN_i[1..N]$ where
$RN_i[j]$ is the largest sequence number received so far in a REQUEST message from site $S_j$. A REQUEST($j$, $n$) message received by site $S_i$ is outdated if $RN_i[j] > n$. When site $S_i$ receives a REQUEST($j$, $n$) message, it sets $RN_i[j] := \max(RN_i[j], n)$.

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 $LN[1..N]$ where $LN[j]$ is the sequence number of the request that 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. The token array $LN[1..N]$ permits a site to determine if some other 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 the token. After having executed 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 ids in queue $Q$ if not already present in this queue $Q$. Then the site sends the token to the site at the head of the queue $Q$.

The Algorithm

Requesting the critical section

1. If the 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 $RN_i[i]$.)
2. 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.

3. Site $S_i$ executes the CS when it has received the token.

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

4. It sets $LN[i]$ element of the token array equal to $RN_i[i]$.
5. For every site $S_j$ whose ID is not in the token queue, it appends its ID to the token queue if $RN_j[j] = LN[j]+1$.
6. If token queue is nonempty after the above update, then it deletes the top site ID from the queue and sends the token to the site indicated by the ID.

Thus, after having executed its CS, a site gives priority to other sites with outstanding requests for the CS (over its pending requests for the CS). The Suzuki-Kasami 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 a symmetric algorithm: “no site possesses the right to access its CS when it has not been requested.”
CORRECTNESS

Theorem 6.5. 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 the 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 execute the CS in finite time. □

PERFORMANCE. The beauty of the Suzuki-Kasami algorithm lies in its simplicity and efficiency. The algorithm requires $O(1)$ or $O(N)$ messages per CS invocation. Synchronization delay in this algorithm is $O(1)$ or $T$. No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request.

6.12 SINGHAL'S HEURISTIC ALGORITHM

In Singhal's token-based heuristic algorithm [20], each site maintains information about the state of other sites in the system and uses it to select a set of sites that are likely to have the token. The site requests the token only from these sites, reducing the number of messages required to execute the CS. It is called a heuristic algorithm because sites are heuristically selected for sending token request messages.

When token request messages are sent only to a subset of sites, it is necessary that a requesting site sends a request message to a site that either holds the token or is going to obtain the token in the near future. Otherwise, there is a potential for deadlocks or starvation. Thus, one design requirement is that a site must select a subset of sites such that at least one of those sites is guaranteed to get the token in near future.

DATA STRUCTURES. A site $S_i$ maintains two arrays, viz., $SV_i[1..N]$ and $SN_i[1..N]$, to store the information about sites in the system. These arrays store the state and the highest known sequence number for each site, respectively. Similarly, the token contains two such arrays as well (denoted by $TSV[1..N]$ and $TSN[1..N]$). Sequence numbers are used to detect outdated requests. A site can be in one of the following states:

- $\mathcal{R}$ - requesting the CS
- $\mathcal{E}$ - executing the CS
- $\mathcal{H}$ - holding the idle token
- $\mathcal{N}$ - none of the above

The arrays are initialized as follows:

For every site $S_i$, $i = 1 \ldots N$ do

\[
\begin{align*}
SV_i[j] &= \mathcal{N} \quad \text{for } j = N \ldots i; \\
SV_i[j] &= \mathcal{R} \quad \text{for } j = i-1 \ldots 1; \\
SN_i[j] &= 0 \quad \text{for } j = 1 \ldots N
\end{align*}
\]
Initially, site \( S_1 \) is in state \( H \) (i.e., \( S_1[1] = H \)).
For the token
\[
\{ TSV[j] = N \text{ and } TSN[j] = 0 \text{ for } j = 1 \ldots N \}
\]

Note that arrays \( SV[1..N] \) of sites are initialized such that for any two sites \( S_i \) and \( S_j \), either \( SV_i[j] = R \) or \( SV_j[i] = R \). Since the heuristic selects every site that is requesting the CS according to local information (i.e., the \( SV \) array), for any two sites that are requesting the CS concurrently, one will always send a token request message to the other. This ensures that sites are not isolated from each other and a site’s request message reaches a site that either holds the token or is going to get the token in near future.

The Algorithm

Requesting the critical section

1. If the requesting site \( S_i \) does not have the token, then it takes the following actions:
   - It sets \( SV_i[i] = R \).
   - It increments \( SN_i[i] := SN_i[i] + 1 \).
   - It sends \( REQUEST(i, sn) \) message to all sites \( S_j \) for which \( SV_i[j] = R \). (\( sn \) is the updated value of \( SN_i[i] \).)

2. When a site \( S_j \) receives the \( REQUEST(i, sn) \) message, it discards the message if \( SN_j[i] \geq sn \) because the message is out dated. Otherwise, it sets \( SN_j[i] \) to \( 'sn' \) and takes the following actions based on its own state:
   - \( SV_j[j] = N \): Set \( SV_j[i] = R \).
   - \( SV_j[j] = R \): If \( SV_j[i] \neq R \), then set \( SV_j[i] := R \) and send a \( REQUEST(j, SN_j[j]) \) message to \( S_i \) (else do nothing).
   - \( SV_j[j] = E \): Set \( SV_j[i] = R \).

Executing the critical section

3. \( S_i \) executes the CS after it has received the token. Before entering the CS, \( S_i \) sets \( SV_i[i] \) to \( E \).

Releasing the critical section

4. Having finished the execution of the CS, site \( S_i \) sets \( SV_i[i] := N \) and \( TSV[i] := N \), and updates its local and token vectors in the following way:

5. If \( (\forall j : SV_j[j] = R) \) then \( SV_i[j] := R \).

The fairness condition states that for every site, if it has sent messages to all the other sites, it is guaranteed that the token reaches a site among the waiting sites. Ideally, if a site is selected to enter its CS, ideally, other sites will not send request messages to cancel its request.

A site updates its local and token vectors for critical section only if it has received a token message from site \( S_i \) which has information about site \( S_i \) holding the token, which has been updated. If at site \( S_i \), \( SN_i[i] \geq sn \), then it does not send request messages to cancel its request if at other sites, \( S_i \) holds the token. Otherwise, it sends request messages to all sites except \( S_j \), otherwise the site \( S_i \) does not send request messages to enter its CS.

CORRECTNESS

Theorem 6.1

Proof: Every site eventually enters the CS. (The proof of this theorem is referred to [2], which is set to R as input.)
For all \( S_j \), \( j = 1 \) to \( N \) do
if \( SN_i[j] > TSN[j] \)
then
(* update token information from local information *)
\{TSV[j] := SV_i[j]; TSN[j] := SN_i[j] \}
else
(* update local information from token information *)
\{SV_i[j] := TSV[j]; SN_i[j] := TSN[j] \}

5. If (\( \forall j :: SV_i[j] = N \)), then set \( SV_i[i] := H \), else send the token to a site \( S_j \) such that \( SV_i[j] = R \).

The fairness of the algorithm depends upon the degree of fairness with which a site is selected for receiving the token, after a site is finished with the execution of its CS. Ideally, the token should not be granted to a site twice or more, while other sites are waiting for the token. Two arbitration rules to ensure fairness in scheduling the token among requesting sites are proposed in [20].

**EXPLANATION.** When a site requests access to the critical section, it sends request messages to all the sites which, according to its local state information, are also currently requesting the CS. The central idea behind the algorithm is that a site's request for the token reaches a site that has the token even though the site does not send request messages to all sites. This is a consequence of the following two factors: (1) How state vectors are initialized and updated and (2) How the sites are selected to send token request messages.

A site updates its state information (arrays \( SN \) and \( SV \)) from the request messages it receives from other sites and from the information in the token that it receives for critical section access. A site \( S_i \) sets \( SV_i[j] \) to \( R \) when it receives a current request message from site \( S_j \) or when it receives the token, which has more up-to-date information about site \( S_j \) and \( TSV[j] := R \). Site \( S_i \) sets \( SV_i[j] \) to \( N \) when it receives the token, which has more up-to-date information about site \( S_j \) and \( TSV[j] := N \). Note that if at site \( S_i \) \( SN_i[j] > TSN[j] \), then site \( S_i \) has more up-to-date information about site \( S_j \), otherwise the token has more up-to-date information about site \( S_j \). Since a site does not send request messages to all sites in the system and a site does not send messages to cancel its request messages, the token plays an important role in the dissemination of system state information.

**CORRECTNESS**

**Theorem 6.6.** A requesting site enters the CS in finite time.

**Proof:** Even though a requesting site does not send token request messages to all other sites, its token request message reaches a site that has the token in finite time. (The proof of this is very complicated and is therefore omitted. Interested readers are referred to [20].) When this site updates the token vector, entry for the requesting site is set to \( R \) and consequently, the requesting site gets the token in finite time. \( \Box \)
PERFORMANCE. A salient feature of this algorithm is that a site can access the critical section without communicating with every site in the system. In low to moderate loads, the average message traffic is \( N/2 \) because each site sends REQUEST messages to half the sites on average. It increases to \( N \) at high loads as most sites will be requesting the CS (which is reflected at site \( S_i \) by \( S_{V_i[j]} = R \) for most \( j \)'s). The synchronization delay in this algorithm is \( T \). An interesting feature of this algorithm is that it adapts itself to the environment of nonuniform traffic of CS requests and to statistical fluctuations in the traffic of CS requests to further reduce the number of messages exchanged.

The algorithm does not have any additional message overhead for the dissemination of state information, except for a slightly larger token message (which is passed comparatively infrequently). Since entries in the token state array (TSV) are either \( R \) or \( N \), this array can be a binary array.

6.13 RAYMOND'S TREE-BASED ALGORITHM

In Raymond's tree-based algorithm [14], sites are logically arranged as a directed tree such that the edges of the tree are assigned directions toward the site (root of the tree) that has the token. Every site has a local variable \( \text{holder} \) that points to an immediate neighbor node on a directed path to the root node. Thus, \( \text{holder} \) variables at the sites define logical tree structure among the sites. If we follow \( \text{holder} \) variables at sites, every site has a directed path leading to the site holding the token. At root site, \( \text{holder} \) points to itself. An example of a tree configuration is shown in Fig. 6.12.

Every site keeps a FIFO queue, called \( \text{request}_q \), which stores the requests of those neighboring sites that have sent a request to this site, but have not yet been sent the token.

The Algorithm

Requesting the critical section

1. When a site wants to enter the CS, it sends a REQUEST message to the node along the directed path to the root, provided it does not hold the token and its \( \text{request}_q \) is empty. If a site is at a site in the top entry

2. When a site \( S_i \) requests the CS, it sends REQUEST messages to the node from which \( S_i \) is a neighbor node on a directed path to the root. Provided it does not hold the token previously

3. When the request message arrives at site \( S_i \), it is placed in the \( \text{request}_q \)

4. When a site \( S_i \) has a REQUEST message to send, the local variable \( \text{holder} \) points to \( S_i \), if it

Executing the critical section

5. A site executes the CS

Releasing the critical section

6. If its \( \text{request}_q \) is empty, it returns the token to the site with the highest \( \text{request}_q \).

7. If the request is not empty, it returns the token to the site with the highest \( \text{request}_q \).

CORRECTNESS

The next show that the tree configuration

Theorem 6.7

Proof: A formal proof is provided in the text.

The execution of requests in its simplest form is that when site \( S_i \) is the root, it requests the CS, which holds the token. When \( S_i \) recognizes that it is the root, it

FIGURE 6.12
Sites arranged in a tree configuration.
can access the low to moderate-level sites will be the number of requests and the number of
or the dissemination of the root site, which is passed
as a directed tree (root of the tree) and its variables at sites, root site, holder
Releasing the critical section. After a site has finished execution of the CS, it takes
the following actions:
6. If its request.q is nonempty, then it deletes the top entry from its request.q, sends
the token to that site, and sets its holder variable to point at that site.
7. If the request.q is nonempty at this point, then the site sends a REQUEST message
to the site which is pointed at by the holder variable.

CORRECTNESS. The algorithm is free from deadlocks because the acyclic nature of
tree configuration eliminates the possibility of circular wait among requesting sites. We
next show that the algorithm is free from starvation.

Theorem 6.7. A requesting site enters the CS in finite time.

Proof: A formal correctness proof is long and complex. Thus, an informal correctness
proof is provided. For a formal proof, readers are referred to the original paper [14].
The essence of proof is based on the following two facts: (1) a site serves requests in its request.q in the FCFS order and (2) every site has a path leading to the site that has the token. Due to the latter fact and Step 2 of the algorithm, when a site $S_i$ is making a request, there exists a chain of requests from site $S_i$ to site $S_h$, which holds the token. Let the chain be denoted by $S_i, S_{i1}, S_{i2}, ..., S_{ik-1}, S_{ik}, S_h$. When $S_h$ receives a REQUEST message from $S_{ik}$, it sends the token to $S_{ik}$. There are two possibilities: $S_{ik-1}$'s request is at the top of $S_{ik}$'s request.q or it is not at
the top. In the first case, $S_{ik}$ sends the token to site $S_{ik-1}$. In the second case, $S_{ik}$ sends the token to the site, say $S_j$, at the top of its request queue and also sends it a REQUEST message (see Step 4 of the algorithm). This extends the chain of requests to $S_i$, $S_{i1}$, $S_{i2}$, ..., $S_{ik-1}$, $S_{ik}$, $S_j$, ..., $S_l$, where site $S_i$ executes the CS next. Note that due to fact (1) above, all the sites in the chain $S_j$, ..., $S_l$ will execute the CS at most once before the token is returned to site $S_{ik}$. Thus, site $S_{ik}$ sends the token to $S_{ik-1}$ in finite time. Likewise, $S_{ik-1}$ sends the token to $S_{ik-2}$ in finite time and so on. Eventually, $S_{i1}$ sends the token to $S_i$. Consequently, a requesting site eventually receives the token. \[ \square \]

**Example 6.4.** In Fig. 6.13, site $S_5$ sends a REQUEST message to $S_2$, which propagates it to the root $S_1$. Root $S_1$ sends the token to $S_2$ which in turn sends the token to $S_5$ (Fig. 6.14). The token travels along the same path traveled by the REQUEST message (but in the opposite direction) and it also reverses the direction of the edges on the path. Consequently, the site that executes the CS last becomes the new root. (See Fig. 6.15.)

![FIGURE 6.13](image)

**FIGURE 6.13**
Site $S_5$ is requesting the token.

![FIGURE 6.14](image)

**FIGURE 6.14**
The token is in transit to $S_5$.

6.14 A COMPARISON OF SYNCHRONIZATION ALGORITHMS

In this section, we compare and contrast algorithms. Table 6.14.1 presents a response time comparison between two site synchronization algorithms. Raymond's algorithm is compared to the algorithm presented in this chapter. Site $S_1$ executes the CS last but note that in both cases, a site only executes the CS if the strategy has an a.

6.14.1 Response Times

At low loads, the response time under a heavy load condition (=$E$). In Raymond's algorithm, the response time as a function of the heavy load condition (=$E$).
The second case, $S_{ik}$ sends the token to $S_k$ and also sends it a chain of requests from the site to the CS next. Note that $S_{ik}$ can execute the CS at $S_k$ because $S_k$ sends the token to itself in a finite time and so an incoming site eventually executes the CS.

In the third case, $S_{ik}$ sends the token to $S_k$, which propagates it to $S_{r_1}$, which then sends the token to $S_{r_2}$ by the REQUEST message, where $r_2$ is the direction of the edges as shown in the figure. It becomes the new root.

**PERFORMANCE.** The average message complexity of Raymond’s algorithm is $O(\log N)$ because the average distance between any two nodes in a tree with $N$ nodes is $O(\log N)$. Synchronization delay in this algorithm is $(T \log N)/2$ because the average distance between two sites to successively execute the CS is $(\log N)/2$.

Raymond’s algorithm can use greedy strategy, where a site receiving the token executes the CS even though its request is not at the top of its request list. It is important to note that in heavy loads, the synchronization delay in this case becomes $T$ because a site executes the CS every time the token is transferred. Needless to say, the greedy strategy has an adverse effect on the fairness of the algorithm and can cause starvation.

### 6.14 A COMPARATIVE PERFORMANCE ANALYSIS

In this section, we present a performance comparison among various mutual exclusion algorithms. Table 6.1 summarizes the response time, the number of messages required, and synchronization delay for mutual exclusion algorithms discussed in this chapter.

#### 6.14.1 Response Time

At low loads, there is hardly any contention among requests for the CS. Therefore, the response time under low load conditions for many algorithms is simply a round trip message delay ($= 2T$) to acquire permission or token plus the time to execute the CS ($= E$). In Raymond’s algorithm, the average distance between a requesting site and the site holding the token is $(\log N)/2$. Thus, the average round trip delay to acquire the token is $T(\log N)$.

As the load is increased, response time increases in all mutual exclusion algorithms because contention for access to the CS increases. Different algorithms see different increases in response time with respect to load. A closed form expression of response time as a function of load is not known for these algorithms. The response time under heavy load conditions is discussed in Sec. 6.14.4 (see Table 6.2 under “Maximum average response time”).
TABLE 6.1
A comparison of performance (ll = light load, hl = heavy load)

<table>
<thead>
<tr>
<th>NON-TOKEN</th>
<th>Resp. time (ll)</th>
<th>Sync Delay</th>
<th>Messages (ll)</th>
<th>Messages (hl)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lamport</td>
<td>$2T+E$</td>
<td>$T$</td>
<td>$3(N - 1)$</td>
<td>$3(N - 1)$</td>
</tr>
<tr>
<td>Ricart-Agrawala</td>
<td>$2T+E$</td>
<td>$T$</td>
<td>$2(N - 1)$</td>
<td>$2(N - 1)$</td>
</tr>
<tr>
<td>Maekawa</td>
<td>$2T+E$</td>
<td>$2T$</td>
<td>$3\sqrt{N}$</td>
<td>$5\sqrt{N}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TOKEN</th>
<th>Resp. time (ll)</th>
<th>Sync Delay</th>
<th>Messages (ll)</th>
<th>Messages (hl)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suzuki and Kasami</td>
<td>$2T+E$</td>
<td>$T$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Singhal’s Heuristic</td>
<td>$2T+E$</td>
<td>$T$</td>
<td>$N/2$</td>
<td>$N$</td>
</tr>
<tr>
<td>Raymond</td>
<td>$T(\log N)+E$</td>
<td>$T\log(N)/2$</td>
<td>$\log(N)$</td>
<td>$4$</td>
</tr>
</tbody>
</table>

6.14.2 Synchronization Delay

Recall from Sec. 6.3.2 that the synchronization delay is the delay due to the sequential message exchanges required after a site leaves the CS and before the next site enters the CS. In many of algorithms, a site exiting the CS directly sends a REPLY message or the token to the next site to enter the CS, resulting in a synchronization delay of $T$. Maekawa’s algorithm has a synchronization delay of $2T$ because a site exiting the CS unlocks an arbiter site (by sending it a RELEASE message) and then the arbiter site sends a GRANT message to the next site to enter the CS (thus, subject to two serial message delays). In Raymond’s algorithm, two sites that consecutively execute the CS can be located at any position in the tree. The token propagates serially along the edges in the tree to the site to enter the CS next. Since the average distance between two nodes in a typical tree with $N$ nodes is $(\log N)/2$, the synchronization delay in this algorithm is $T(\log N)/2$. However, when structure-based token algorithms (such as Raymond [14]) use a greedy strategy, synchronization delay in heavy loads becomes $T$.

6.14.3 Message Traffic

The Lamport’s, Ricart-Agrawala, and Suzuki-Kasami algorithms respectively require $3*(N - 1)$, $2*(N - 1)$, and $N$ messages per CS execution, irrespective of the load. In other algorithms, the number of messages needed depends upon the load and is discussed below:

**Light Load.** Raymond’s algorithm requires $\log N$ messages per CS execution. This is because a request for a token must travel along the edges of the tree from the requester node to the root node and the token must travel back from the root node to the requester node. Note that the average distance between the root and a node in a typical tree with $N$ nodes is $(\log N)/2$. Singhal’s heuristic algorithm requires $N/2$ message per CS execution under light load conditions.

**Heavy Load.** In Raymond’s algorithm, message traffic decreases as load increases due to the sharing of request messages at high loads—when a site receives a request message from an predecessor site. Under heavy load, it can be seen that the synchronization delay becomes $T$. When the site has performed $T$ operations in the CS, it sends a REPLY message to the next site to enter the CS, resulting in a message delay of $T$. In general, the message delay is $T$ for most algorithms under heavy load conditions.

LOAD NONUNIF

requests is nonuniform, indicating the potential to substantial optimization. Likewise, if the number of messages in Raymond’s algorithms is $N$, then the algorithm under heavy load conditions.

6.14.4 Universal Metrics

Due to their idiosyncrasies, the algorithms are not compared based on their performance characteristics, but upon any of these bounds.

6.15 SUMMARY

The problem of designing a mutual exclusion algorithm under the last decade. In this section, we have discussed the various aspects of mutual exclusion algorithms, focusing on their performance characteristics under different load conditions. We have also presented a comparison table that provides a comprehensive view of the algorithms, highlighting their synchronization delays and message traffic.

TABLE 6.2
Universal performance

<table>
<thead>
<tr>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum synchronization delay</td>
</tr>
<tr>
<td>Maximum throughput</td>
</tr>
<tr>
<td>Minimum response time</td>
</tr>
<tr>
<td>Maximum average response time</td>
</tr>
</tbody>
</table>
message from another site, it does not forward it if it has already sent a request message. Under heavy load conditions, message traffic in Singhal’s heuristic algorithm degenerates to that in the Suzuki-Kasami algorithm and requires $N$ messages per CS execution.

LOAD NONUNIFORMLY DISTRIBUTED OVER SITES. When the rate of CS requests is nonuniformly distributed over sites, Singhal’s heuristic algorithm [20] has the potential to substantially reduce the message traffic by adapting to the prevailing system state. Likewise, piggybacking and greedy strategies can substantially reduce message traffic in Raymond’s algorithm [14] provided requests for the CS are localized to a few neighboring sites. For example, with these strategies, the message traffic in Raymond’s algorithm under heavy load conditions is four messages per CS execution.

6.14.4 Universal Performance Bounds

Due to their idiosyncrasies, different mutual exclusion algorithms have widely varying performance characteristics. Nevertheless, we can determine universal performance bounds for mutual exclusion algorithms. These bounds depend upon system characteristics, not upon any particular mutual exclusion algorithm. In Table 6.2, we summarize these bounds.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Bound</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum synchronization</td>
<td>$T$</td>
<td>At least one message has to be transferred before the next site can receive the notification of permission to enter the CS.</td>
</tr>
<tr>
<td>delay</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum throughput</td>
<td>$1/(T+E)$</td>
<td>The minimum time interval between two successive CS execution is $T+E$.</td>
</tr>
<tr>
<td>Minimum response time</td>
<td>$2T+E$</td>
<td>One round trip message delay to acquire permission or the token plus time to execute the CS.</td>
</tr>
<tr>
<td>Maximum average</td>
<td>$N(T+E)$</td>
<td>This occurs at heavy loads (when all sites always have pending requests).</td>
</tr>
<tr>
<td>response time</td>
<td></td>
<td>Note that system executes requests at a rate $r = 1/(T+E)$. From Little’s Law, response time $= N/r = N(T+E)$.</td>
</tr>
</tbody>
</table>

TABLE 6.2

Universal performance bounds

6.15 SUMMARY

The problem of distributed mutual exclusion has received considerable attention over the last decade. In this chapter, a common framework has been provided with which to
analyze several significant distributed mutual exclusion algorithms. These algorithms have been summarized and their features and performance compared.

Token-based mutual exclusion algorithms are in general more message-efficient than non-token-based mutual exclusion algorithms—a single token message is returned instead of individual reply messages from a number of sites. Thus, the token can be viewed as all the reply messages lumped together. While a general theory exists to unify all non-token-based mutual exclusion algorithms, no such unifying theme exists for token-based mutual exclusion algorithms. It has been found that, in general, algorithms that make decisions based on the current state of the system (e.g., [20]) are more efficient. Universal performance bounds (due to physical limitations of the system parameters) for mutual exclusion algorithms have been discussed.

In general, there is a tradeoff between speed and message complexity in mutual exclusion algorithms. There is no single algorithm that can optimize both speed and message complexity. Future research on the design of efficient mutual exclusion algorithms should focus on hybrid mutual exclusion algorithms that combine the advantages of two mutual exclusion algorithms to provide improved performance in both speed and message complexity.

### 6.16 FURTHER READING

In recent years, there has been burgeoning growth in the literature on distributed mutual exclusion. Taxonomy on distributed mutual exclusion can be found in [15] and [18]. A large number of structure-based token algorithms have appeared in the last several years. Distinct examples are mutual exclusion algorithms by Bernabeu-Auban and Ahamad [2], Helary et al. [7], Naimi and Trehel [11], and Neilsen and Mizuno [12]. Nishio et al. [13] present a technique for the generation of a unique token in case of a token loss. Van-de-Snepscheut [22] extends tree-based algorithms to handle a connected network of any topology (i.e., graphs). Due to network topology, such an algorithm is fault-tolerant to site and link failures. Goscinski [6] 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 attracted considerable attention of late. Barbara and Garcia-Molina [5] and Ibaraki and Kameda [8] have discussed theoretical aspects of coteries. Agrawal and El Abbadi [1] presented a coterie algorithm for mutual exclusion that generates coteries from a tree configuration of sites.

### PROBLEMS

6.1. In Lamport’s algorithm, condition L1 can hold concurrently at several sites. Why then is L1 needed to guarantee mutual exclusion?

6.2. 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?
6.3. Maekawa’s mutual exclusion algorithm gives the impression that message complexity of a distributed mutual exclusion can be $O(\sqrt{N})$ instead of $O(N)$, as in many other mutual exclusion algorithms. Discuss how Maekawa’s algorithm fundamentally differs from other algorithms and what problems it poses.

6.4. What is the purpose of a REPLY message in Lamport’s algorithm? Note that a site need not necessarily return a REPLY message in response to a REQUEST message. State the condition under which a site does not have to return a REPLY message. Also, give the new message complexity per critical section execution in this case.

6.5. Show that in the Ricart-Agrawala algorithm, the critical section is accessed according to the increasing order of timestamps. Does the same hold true in Maekawa’s algorithm?

6.6. Mutual exclusion can be achieved in the following simple way in a distributed system (called the centered mutual exclusion algorithm): Every site, when it receives a request to access the shared resource from users, sends the request to the site which contains the resource. This site executes the requests using any classical method for mutual exclusion (e.g., semaphores). Discuss what prompted Lamport’s mutual exclusion algorithm even though it requires many more messages ($3(N-1)$ as compared to only 3).

6.7. Show that in Lamport’s algorithm the critical section is accessed according to the increasing order of timestamps.

REFERENCES


7.1 INTRODUCTION
A distributed system is composed of local users connected by message passing to local users and requesting a service. The request and release of a resource can be made a priori and a posteriori of the allocation of resources. Deadlocks can occur in distributed systems. Several models are described. In Chap. 4, the model is compared.

7.2 PRELIMINARIES
7.2.1 The System
In Chap. 3, the parallel and distributed model was described. Various special systems are studied in distribute...
7.1 INTRODUCTION

A distributed system is a network of sites that exchange information with each other by message passing. A site consists of computing and storage facilities and interface to local users and a communication network. In distributed systems, a process can request and release resources (local or remote) in any order, which may not be known a priori and a process can request some resources while holding others. If the sequence of the allocation of resources to processes is not controlled in such environments, deadlocks can occur. In this chapter, we study deadlock handling strategies in distributed systems. Several deadlock detection techniques based on various control organizations are described. Pros and cons of these techniques are discussed and their performance is compared.

7.2 PRELIMINARIES

7.2.1 The System Model

In Chap. 3, the problem of deadlocks in centralized systems under a very general system model was discussed. Conditions for deadlocks and methods to handle deadlocks in various special cases were discussed. The problem of deadlocks has been generally studied in distributed systems under the following model:
The systems have only reusable resources.
- Processes are allowed only exclusive access to resources.
- There is only one copy of each resource.

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

### 7.2.2 Resource vs. Communication Deadlocks

Two types of deadlocks have been discussed in the literature: resource deadlock and communication deadlock. In resource deadlocks, processes can simultaneously wait for several resources and cannot proceed until they have acquired all those resources. A set of processes is resource-deadlocked if each process in the set requests resources held by another process in the set and it must receive all of the requested resources before it can become unblocked.

In communication deadlocks [5], processes wait to communicate with other processes among a set of processes. A waiting process can unblock on receiving a communication from any one of these processes. A set of processes is communication-deadlocked if each process in the set is waiting to communicate with another process in the set and no process in the set ever initiates any further communication until it receives the communication for which it is waiting. Note that a “wait to communicate” can be viewed as a “wait to acquire a resource”. Thus, the communication model is the same as the OR request model discussed in Chap. 3.

### 7.2.3 A Graph-Theoretic Model

As in Chap. 3, the state of process-resource interaction in distributed systems can be modeled by a bi-partite directed graph called a resource allocation graph. The nodes of this graph are processes and resources of a system, and the edges of the graph depict assignments or pending requests. A pending request is represented by a request edge directed from the node of a requesting process to the node of the requested resource. A resource assignment is represented by an assignment edge directed from the node of an assigned resource to the node of the assigned process. A system is deadlocked if its resource allocation graph contains a directed cycle or a knot. (A knot is defined in Chap. 3.)

### Wait-For Graphs

In distributed systems, the system state can be modeled or represented by a directed graph, called a wait-for graph (WFG). In a WFG, nodes are processes and there is a directed edge from node \( P_1 \) to node \( P_2 \) if \( P_1 \) is blocked and is waiting for \( P_2 \) to release some resource. A system is deadlocked if and only if there is a directed cycle or knot (depending upon the underlying model) in the WFG.

In distributed databases by exchanging transactions, a database can be (through unlock) to as a transaction transactions and if is waiting for \( T_2 \) is a directed cycle.
In distributed database systems (DDBS), users access the data objects of the database by executing transactions. A transaction can be viewed as a process that performs a sequence of reads and writes on the data objects. The data objects of a database can be viewed as resources that are acquired (through locking) and released (through unlocking) by transactions. In DDBS literature, a wait-for graph is referred to as a transaction-wait-for graph (TWF graph) [22]. In a TWF graph, nodes are transactions and there is a directed edge from node $T_1$ to node $T_2$ if $T_1$ is blocked and is waiting for $T_2$ to release some resource. A system is deadlocked if and only if there is a directed cycle or a knot in its TWF graph.

7.3 DEADLOCK HANDLING STRATEGIES IN DISTRIBUTED SYSTEMS

Recall from Chap. 3 that there are three strategies to handle deadlocks, viz., deadlock prevention, deadlock avoidance, and deadlock detection. Note that deadlock handling is complicated to implement in distributed systems because no one site has accurate knowledge of the current state of the system and because every intersite communication involves a finite and unpredictable delay. An examination of the relative complexity and practicality of these three deadlock handling strategies in distributed systems follows.

7.3.1 Deadlock Prevention

Deadlock prevention is commonly achieved by either having a process acquire all the needed resources simultaneously before it begins execution or by preempting a process that holds the needed resource. In the former method, a process requests (or releases) a remote resource by sending a request message (or release message) to the site where the resource is located. This method has a number of drawbacks. First, it is inefficient as it decreases the system concurrency. Second, a set of processes can become deadlocked in the resource acquiring phase. For example, suppose process $P_1$ at site $S_1$ and process $P_2$ at site $S_2$ simultaneously request two resources $R_3$ and $R_4$, located at sites $S_3$ and $S_4$, respectively. It may happen that $S_3$ grants $R_3$ to $P_1$ and $S_4$ grants $R_4$ to $P_2$, resulting in a deadlock. This problem can be handled by forcing processes to acquire needed resources one by one; however, this approach is highly inefficient and impractical. Third, in many systems, future resource requirements are unpredictable (i.e., not known a priori).

7.3.2 Deadlock Avoidance

In the deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system). Because of the following problems, deadlock avoidance can be seen as impractical in distributed systems: (1) Every site has to maintain information on the global state of the system, which translates into huge storage requirements and extensive communication costs (as every change in the global state has to be communicated to every site). (2) The process of checking for a safe
global state must be mutually exclusive, because if several sites concurrently perform checks for a safe global state (each site for a different resource request), they may all find the state safe but the net global state may not be safe. This restriction will severely limit the concurrency and throughput of the system. (3) Due to the large number of processes and resources, it will be computationally expensive to check for a safe state.

7.3.3 Deadlock Detection

Deadlock detection requires an examination of the status of process-resource interactions for the presence of cyclical wait. Deadlock detection in distributed systems has two favorable conditions: (1) Once a cycle is formed in the WFG, it persists until it is detected and broken and (2) cycle detection can proceed concurrently with the normal activities of a system (and therefore it does not have a negative effect on system throughput). Because of this, the literature on deadlock handling in distributed systems is highly focused toward deadlock detection methods. In this chapter, the discussion is limited to deadlock detection techniques in distributed systems.

7.4 ISSUES IN DEADLOCK DETECTION AND RESOLUTION

Deadlock detection and resolution entails addressing two basic issues: First, detection of existing deadlocks and second resolution of detected deadlocks.

Detection

The detection of deadlocks involves two issues: maintenance of the WFG and search of the WFG for the presence of cycles (or knots). In distributed systems, a cycle may involve several sites, so the search for cycles greatly depends upon how the WFG of the system is represented across the system. Depending upon the manner in which WFG information is maintained and the search for cycles is carried out, there are centralized, distributed, and hierarchical algorithms for deadlock detection in distributed systems [27]. We discuss these control organizations in detail in the next section.

A correct deadlock detection algorithm must satisfy the following two conditions:

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

- **Safety—No false deadlocks.** The algorithm should not report deadlocks which are non-existent (called phantom deadlocks). In distributed systems where there is no global memory and communication occurs solely by messages, it is difficult to design a correct deadlock detection algorithm because sites may obtain out of date and inconsistent WFGs of the system. As a result, sites may detect a cycle that doesn’t exist, but whose different segments were existing in the system at different times. This is the primary reason why many deadlock detection algorithms reported in the literature are incorrect.

Resolution

Deadlock resolution requires an examination of the status of process-resource interactions for the presence of cyclical wait. Deadlock detection in distributed systems has two favorable conditions: (1) Once a cycle is formed in the WFG, it persists until it is detected and broken and (2) cycle detection can proceed concurrently with the normal activities of a system (and therefore it does not have a negative effect on system throughput). Because of this, the literature on deadlock handling in distributed systems is highly focused toward deadlock detection methods. In this chapter, the discussion is limited to deadlock detection techniques in distributed systems.

7.5 CONTROL ORGANIZATIONS FOR DEADLOCK DETECTION

7.5.1 Centralized

In centralized deadlock detection, the system has a control site (or server or central processor) that monitors the status of all process-resource interactions in the system. The control site maintains the WFG of the system and, when a deadlock is detected, it can immediately resolve the deadlock. However, communication is required between the sites and the control site, which may not be feasible in large distributed systems.

7.5.2 Distributed

In distributed deadlock detection, the WFG is maintained at each site. When a deadlock is detected, the site initiates deadlock resolution. However, since the WFG is decentralized, it can be very difficult to resolve a deadlock when the sites are geographically dispersed and communication is expensive.
Currently perform (rce interactions (rce), they may all (rce). If (rce) will severely (rce) a large number of (rce), it may be necessary for a safe state.

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

7.5 CONTROL ORGANIZATIONS FOR DISTRIBUTED DEADLOCK DETECTION

7.5.1 Centralized Control

In centralized deadlock detection algorithms, a designated site (often called a control site) has the responsibility of constructing the global WFG and searching it for cycles. The control site may maintain the global WFG constantly or it may build it whenever a deadlock detection is to be carried out by soliciting the local WFG from every site. Centralized deadlock detection algorithms are conceptually simple and are easy to implement. Deadlock resolution is simple in these algorithms—the control site has complete information about the deadlock cycle and it can thus optimally resolve the deadlock.

However, centralized deadlock-detection algorithms have a single point of failure. Communication links near the control site are likely to be congested because the control site receives WFG information from all other sites. Also, the message traffic generated due to deadlock detection activity is independent of the rate of deadlock formation and the structure of deadlock cycles.

7.5.2 Distributed Control

In distributed deadlock detection algorithms, the responsibility for detecting a global deadlock is shared equally among all sites. The global state graph is spread over many sites and several sites participate in the detection of a global cycle. Unlike centralized control, distributed control is not vulnerable to a single point of failure and no site is swamped with deadlock detection activity. Also, a deadlock detection is initiated only when a waiting process is suspected to be a part of a deadlock cycle.

Distributed deadlock detection algorithms are difficult to design due to the lack of globally shared memory—sites may collectively report the existence of a global cycle after seeing its segments at different instants (though all the segments never existed simultaneously). Unlike centralized control, several sites in distributed control may initiate deadlock detection for the same deadlock. Also, the proof of correctness is difficult for these algorithms. In addition, deadlock resolution is often cumbersome in distributed deadlock detection algorithms, as several sites can detect the same deadlock and not be aware of the other sites or processes involved in the deadlock.
7.5.3 Hierarchical Control

In hierarchical deadlock detection algorithms, sites are arranged in a hierarchical fashion, and a site detects deadlocks involving only its descendant sites. Hierarchical algorithms exploit access patterns local to a cluster of sites to efficiently detect deadlocks. They tend to get the best of both the centralized and the distributed control organizations in that there is no single point of failure (as in centralized control) and a site is not bogged down by deadlock detection activities with which it is not concerned (as sometimes happens in distributed control). However, hierarchical deadlock detection algorithms require special care while arranging the sites in a hierarchy. For efficiency, most deadlocks should be localized to as few clusters as possible—the objective of hierarchical control is defeated if most deadlocks span several clusters.

A description of deadlock detection algorithms, based on the centralized, distributed, and hierarchical control organizations follows. For these algorithms, we describe the basic ideas behind their operation, compare them with each other, and discuss their pros and cons. We also summarize the performance of these algorithms in terms of message traffic, message size, and delay in detecting a deadlock. Due to the following reasons, it is not possible to enumerate these performance measures with high accuracy for many deadlock detection algorithms: the statistical nature of the topology of WFG graph, the invocation of deadlock detection activities despite the absence of a deadlock, the initiation of detection of a deadlock by several processes in a deadlock cycle, etc. Therefore, for most algorithms performance bounds (e.g., the maximum number of messages transferred to detect a global cycle) rather than exact numbers are used as a means of comparison and performance analysis.

7.6 CENTRALIZED DEADLOCK-DETECTION ALGORITHMS

7.6.1 The Completely Centralized Algorithm

The completely centralized algorithm is the simplest centralized deadlock detection algorithm, wherein a designated site called the control site, maintains the WFG of the entire system and checks it for the existence of deadlock cycles. All sites request and release resources (even local resources) by sending request resource and release resource messages to the control site, respectively. When the control site receives a request resource or a release resource message, it correspondingly updates its WFG. The control site checks the WFG for deadlocks whenever a request edge is added to the WFG.

This algorithm is conceptually simple and easy to implement. Unfortunately, it is also highly inefficient because all resource acquisition and release requests must go through the control site, even when the resource is local. This results in large delays in responding to user requests, large communication overhead, and the congestion of communication links near the control site. Moreover, reliability is poor because if the control site fails, the entire system comes to a halt because all the status information resides at the control site.

Several problems related to the centralization of communication exist. Sites must maintain its resource status to a designated site to detect a deadlock. Even though the lack of perfectly consistent algorithm of the system among the sites is an issue.

For example, suppose that the lock(R) request of T1 to a designated site, S1, fails, and the request to S2, T2, succeeds. The lock(R2) request of T1 to S2 fails. The lock(R2) request of T1 which waits for a lock(R2) granted by S1. Therefore, for most algorithms performance bounds (e.g., the maximum number of messages transferred to detect a global cycle) rather than exact numbers are used as a means of comparison and performance analysis.

7.6.2 The Ho-Ramamoorthi Algorithm

Ho and Ramamoorthi [1990] developed the two-phase algorithm, which consists of two algorithms, respectively. Suppose that the lock(R1) request of T1 to a designated site, S1, fails, and the lock(R2) request of T1 to S2, T2, succeeds. Periodically, a designated site, S1, from the information of the system is free from all the sites and from the lock(R1) request of T1 which waits for a lock(R1) granted by S1. Therefore, for most algorithms performance bounds (e.g., the maximum number of messages transferred to detect a global cycle) rather than exact numbers are used as a means of comparison and performance analysis.

It was claimed that the consecutive reports, respectively, is consistent if it reflects
Hierarchical fashion, a hierarchical algorithm can detect deadlocks. For control organization (global control) and a site is not concerned (as in the case of deadlock detection algorithms), for a site. For efficiency, the objective of the algorithm is to minimize the number of messages.

For centralized, distributed, and control algorithms, we describe them further, and discuss them in terms of the following high accuracy topology of WFG.

The presence of a deadlock cycle, minimum number of transactions are used as a strategy.

7.6.2 The Ho-Ramamoorthy Algorithms

Ho and Ramamoorthy gave two centralized deadlock detection algorithms, called two-phase and one-phase algorithms [14], to fix the problem of the above algorithm. These algorithms, respectively, collect two consecutive status reports or keep two status tables at each site to ensure that the control site gets a consistent view of the system.

THE TWO-PHASE ALGORITHM. In the two-phase algorithm, every site maintains a status table that contains the status of all the processes initiated at that site. The status of a process includes all resources locked and all resources being waited upon. Periodically, a designated site requests the status table from all sites, constructs a WFG from the information received, and searches it for cycles. If there is no cycle, then the system is free from deadlocks, otherwise, the designated site again requests status tables from all the sites and again constructs a WFG using only those transactions which are common to both reports. If the same cycle is detected again, the system is declared deadlocked.

It was claimed that by selecting only the common transactions found in two consecutive reports, the algorithm gets a consistent view of the system. (A view is consistent if it reflects a correct state of the system.) If a deadlock exists, it was argued,
the same wait-for condition must exist in both reports. However, this claim proved to be incorrect (i.e., a cycle in the wait-for conditions of the transactions common in two consecutive reports does not imply a deadlock) and two-phase algorithm may indeed report false deadlocks. By getting two consecutive reports, the designated site reduces the probability of getting an inconsistent view, but does not eliminate such a possibility.

THE ONE-PHASE ALGORITHM. The one-phase algorithm requires only one status report from each site; however, each site maintains two status tables: a resource status table and a process status table. The resource status table at a site keeps track of the transactions that have locked or are waiting for resources stored at that site. The process status table at a site keeps track of the resources locked by or waited for by all the transactions at that site. Periodically, a designated site requests both the tables from every site, constructs a WFG using only those transactions for which the entry in the resource table matches the corresponding entry in the process table, and searches the WFG for cycles. If no cycle is found, then the system is not deadlocked, otherwise a deadlock is detected.

The one-phase algorithm does not detect false deadlocks because it eliminates the inconsistency in state information by using only the information that is common to both tables. This eliminates inconsistencies introduced by unpredictable message delays. For example, if the resource table at site $S_1$ indicates that resource $R_1$ is waited upon by a process $P_2$ (i.e., $R_1 \rightarrow P_2$) and the process table at site $S_2$ indicates that process $P_2$ is waiting for resource $R_1$ (i.e., $P_2 \rightarrow R_1$), then edge $P_2 \rightarrow R_1$ in the constructed WFG reflects the correct system state. If either of these entries is missing from the resource or the process table, then a request message or a release message from $S_2$ to $S_1$ is in transit and $P_2 \rightarrow R_1$ cannot be ascertained.

The one-phase algorithm is faster and requires fewer messages as compared to the two-phase algorithm. However, it requires more storage because every site maintains two status tables and exchanges bigger messages because a message contains two tables instead of one.

7.7 DISTRIBUTED DEADLOCK DETECTION ALGORITHMS

In distributed deadlock detection algorithms, all sites collectively cooperate to detect a cycle in the state graph that is likely to be distributed over several sites of the system. A distributed deadlock detection algorithm can be initiated whenever a process is forced to wait, and it can be initiated either by the local site of the process or by the site where the process waits.

Distributed deadlock detection algorithms can be divided into four classes [18]: path-pushing, edge-chasing, diffusion computation, and global state detection.

In path-pushing algorithms, the wait-for dependency information of the global WFG is disseminated in the form of paths (i.e., a sequence of wait-for dependency edges). Classic examples of such algorithms are Menasce-Muntz [22] and Obermarck [24] algorithms.
In edge-chasing algorithms, special messages called probes are circulated along the edges of the WFG to detect a cycle. When a blocked process receives a probe, it propagates the probe along its outgoing edges in the WFG. A process declares a deadlock when it receives a probe initiated by it. An interesting feature of edge-chasing algorithms is that probes are of a fixed size (normally very short). Examples of these algorithms include Chandy et al. [5] and Sinha-Natarajan [28] algorithms.

Diffusion computation type algorithms make use of echo algorithms to detect deadlocks [6]. To detect a deadlock, a process sends out query messages along all the outgoing edges in the WFG. These queries are successively propagated through the edges of the WFG. (As the name implies, these queries are diffused through the WFG.) Queries are discarded by a running process and are echoed back by blocked processes in the following way: When a blocked process receives first query message for a particular deadlock detection initiation, it does not send a reply message until it has received a reply message for every query it sent (to its successors in the WFG). For all subsequent queries for this deadlock detection initiation, it immediately sends back a reply message. The initiator of a deadlock detection detects a deadlock when it receives a reply for every query it sent. Some examples of these types of deadlock detection algorithms are Chandy-Misra-Haas algorithm for the OR request model [5] and Chandy-Herman algorithm [13].

Global state detection based deadlock detection algorithms exploit the following facts:

- A consistent snapshot of a distributed system can be obtained without freezing the underlying computation.
- A consistent snapshot may not represent the system state at any moment in time, but if a stable property holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot.

Therefore, distributed deadlocks can be detected by taking a snapshot of the system and examining it for the condition of a deadlock. Examples of these types of algorithms include Bracha-Toueg [2], Wang et al. [30], and Kshemkalyani-Singhal [20] algorithms.

### 7.7.1 A Path-Pushing Algorithm

In path-pushing deadlock detection algorithms, information about the wait-for dependencies is propagated in the form of paths. Obermarck’s algorithm [24] is chosen to illustrate a path-pushing deadlock detection algorithm because it is implemented on the distributed database system R* of the IBM Corporation.

Obermarck’s algorithm [24] was designed for distributed database systems; therefore, processes are referred to as transactions which are denoted by $T_1, T_2, ..., T_n$. A transaction may consist of several subtransactions that normally execute at different sites. Obermarck’s model assumes that at most one subtransaction within a given transaction can be executing at a time. Execution sequentially passes from subtransaction to subtransaction. The subtransactions communicate synchronously by passing messages.
Obermarck’s algorithm has two interesting features:

- The nonlocal portion of the global TWF graph at a site is abstracted by a distinguished node (called External or Ex) which helps in determining potential multisite deadlocks without requiring a huge global TWF graph to be stored at each site.
- Transactions are totally ordered, which reduces the number of messages and consequently decreases deadlock detection overhead. It also ensures that exactly one transaction in each cycle detects the deadlock.

THE ALGORITHM. Deadlock detection at a site follows the following iterative process:

1. The site waits for deadlock-related information (produced in Step 3 of the previous deadlock detection iteration) from other sites. (Note that deadlock-related information is passed by sites in the form of paths.)
2. The site combines the received information with its local TWF graph to build an updated TWF graph. It then detects all cycles and breaks only those cycles which do not contain the node ‘Ex’. Note that these cycles are local to this site. All other cycles have the potential to be a part of global cycles.
3. For all cycles ‘Ex → T₁ → T₂ → Ex’ which contain the node ‘Ex’ (these cycles are potential candidates for global deadlocks), the site transmits them in string form ‘Ex, T₁, T₂, Ex’ to all other sites where a subtransaction of T₂ is waiting to receive a message from the subtransaction of T₂ at this site. The algorithm reduces message traffic by lexically ordering transactions and sending the string ‘Ex, T₁, T₂, T₃, Ex’ to other sites only if T₁ is higher than T₃ in the lexical ordering. Also, for a deadlock, the highest priority transaction detects the deadlock.

Obermarck gave an informal correctness proof of the algorithm [24]. However, the algorithm is incorrect because it detects phantom deadlocks. The main reason for this is that the portions of TWF graphs that are propagated to other sites may not represent a consistent view of the global TWF graph. This is because each site takes its snapshot asynchronously at Step 2. Consequently, when a site sends out portions of its TWF graph as paths to other sites in Step 3, the global dependency represented by this path may change without this site knowing about it.

This algorithm sends \( n(n - 1)/2 \) messages to detect a deadlock involving \( n \) sites. Size of a message is \( O(n) \). The delay in detecting the deadlock is \( O(n) \).

7.7.2 An Edge-Chasing Algorithm

We discuss Chandy-Misra-Haas’s distributed deadlock detection algorithm [5] for the AND request model to illustrate deadlock detection using edge-chasing.

Chandy et al.’s algorithm [5] uses a special message called a probe. A probe is a triplet \((i, j, k)\) denoting that it belongs to a deadlock detection initiated for process \(P_i\) and it is being sent by the home site of process \(P_j\) to the home site of process \(P_k\).

A probe message is detected when a site \(P_i\) receives a probe message \((i, j, k)\) with \(j \neq \) home site and \(k \neq \) home site of \(P_i\).

We now describe the distributed algorithm using an example.

Example 7.1. Suppose processes \(P_{11}, P_{12}, ..., P_{1m}\) are arranged in a linear sequence, where \(P_{1j}\) is at position \(j\). Each process \(P_i\) knows that \(P_{i+1}\) is at position \(i+1\) in the sequence, and \(P_{i+1}\) is the successor of \(P_i\). On the other hand, \(P_i\) depends upon \(P_{i-1}\). Thus, a probe message \((i, j, k)\) is sent by process \(P_j\) and arrives at a process \(P_i\) where \(j < i < k\).

On the receipt of a probe message, \(P_i\) initiates deadlock detection. Suppose \(P_i\) is the home site of process \(P_i\) and \(P_j\) is waiting for a message from \(P_i\).
A probe message travels along the edges of the global TWF graph, and a deadlock is detected when a probe message returns to its initiating process.

We now define terms and data structures used in the algorithm. A process $P_j$ is said to be dependent on another process $P_k$ if there exists a sequence of processes $P_j, P_{i1}, P_{i2}, ..., P_{im}, P_k$ such that each process except $P_k$ in the sequence is blocked and each process, except the first one ($P_j$), holds a resource for which the previous process in the sequence is waiting. Process $P_j$ is locally dependent upon process $P_k$ if $P_j$ is dependent upon $P_k$ and both the processes are at the same site. The system maintains a boolean array, dependent$_i$ for each process $P_i$, where dependent$_i$(j) is true only if $P_i$ knows that $P_j$ is dependent on it. Initially, dependent$_i$(j) is false for all $i$ and $j$.

**THE ALGORITHM.** To determine if a blocked process is deadlocked, the system executes the following algorithm:

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

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

```
if
    (d) $P_k$ is blocked, and
    (e) dependent$_k(i)$ is false, and
    (f) $P_k$ has not replied to all requests of $P_j$,
then
    begin
        dependent$_k(i) = $true;
        if $k = i$
            then declare that $P_i$ is deadlocked
        else for all $P_m$ and $P_n$ such that
            (a') $P_k$ is locally dependent upon $P_m$, and
            (b') $P_m$ is waiting on $P_n$, and
            (c') $P_m$ and $P_n$ are on different sites,
            send probe $(i, m, n)$ to the home site of $P_n$
        end.
```

Thus, a probe message is successively propagated along the edges of the global TWF graph and a deadlock is detected when a probe message returns to its initiating process.

**Example 7.1.** As an example, consider the system shown in Fig. 7.1. If process $P_1$ initiates deadlock detection, it sends probe $(1, 3, 4)$ to $S_2$. Since $P_3$ is waiting for $P_4$ and $P_1$ is waiting for $P_{10}$, $S_2$ sends probes $(1, 6, 8)$ and $(1, 7, 10)$ to $S_3$ which in
FIGURE 7.1
An example of Chandy et al.'s edge-chasing algorithm.

Chandy et al.'s algorithm sends one probe message (per deadlock detection initiation) on each edge of the WFG, which spans two sites. Thus, the algorithm at most exchanges \( m(n - 1)/2 \) messages to detect a deadlock that involves \( m \) processes and spans over \( n \) sites. The size of messages exchanged is fixed and very small (only 3 integer words). The delay in detecting the deadlock is \( O(n) \).

OTHER EDGE-CHASING ALGORITHMS

The Mitchell-Merritt Algorithm. In the deadlock detection algorithm of Mitchell and Merritt [23], each node of the TWF graph has two labels: private and public. The private label of each node is unique to that node, and initially both labels at a node have the same value. The algorithm detects a deadlock by propagating the public label of nodes in the backward direction in the TWF graph. When a transaction is blocked, the public and private label of its node in the TWF graph are changed to a value greater than their previous values and greater than the public label of the blocking transaction. A blocked transaction periodically reads the public label of the blocking transaction and replaces its own public label with it, provided the blocking transaction's public label is larger than its own. A deadlock is detected when a transaction receives its own public label. In essence, the largest public label propagates in the backward direction in a deadlock cycle. Deadlock resolution is simple in this algorithm because only one process detects a deadlock and that process can resolve the deadlock by simply aborting itself.
Sinha-Natarajan Algorithm. In the Sinha-Natarajan algorithm [28], transactions are assigned unique priorities, and an antagonistic conflict is said to occur when a transaction waits for a data object that is locked by a lower priority transaction. The algorithm initiates deadlock detection only when an antagonistic conflict occurs.

The algorithm detects a deadlock by circulating a probe message through a cycle in the global TWF graph. A probe message is a 2-tuple \((i, j)\) where \(T_i\) is the transaction that initiated the deadlock detection and \(T_j\) is the transaction whose priority is the lowest among all the transactions (i.e., nodes of the TWF graph) the probe has traversed so far. When a waiting transaction receives a probe that was initiated by a lower priority transaction, the probe is discarded.

An interesting property of this algorithm is that a deadlock is detected when the probe issued by the highest priority transaction in the cycle returns to it. (There is only one detector of every deadlock.) Deadlock resolution is simple because the detector of a deadlock can resolve the deadlock by aborting the lowest priority transaction of the cycle. This was the first algorithm to comprehensively treat deadlock resolution.

Choudhary et al. showed that the Sinha-Natarajan algorithm detects false deadlocks and fails to report all deadlocks because it overlooks the possibility of a transaction waiting transitively on a deadlock cycle and because the probes of aborted transactions are not cleaned properly [7]. Choudhary et al. proposed a corrected version of the Sinha-Natarajan algorithm, but it has been shown that Choudhary et al.'s corrected algorithm still detects false deadlocks and fails to report all deadlocks [19].

7.7.3 A Diffusion Computation Based Algorithm

In diffusion computation based distributed deadlock detection algorithms, deadlock detection computation is diffused through the WFG of the system. Chandy et al.'s distributed deadlock detection algorithm for the OR request model [5] is discussed to illustrate the technique of diffusion computation based algorithms.

A process determines if it is deadlocked by initiating a diffusion computation. The messages used in diffusion computation take the form of a query\((i, j, k)\) and a reply\((i, j, k)\), denoting that they belong to a diffusion computation initiated by a process \(P_i\) and are being sent from process \(P_j\) to process \(P_k\). A process can be in two states: active or blocked. In the active state, a process is executing and in the blocked state, a process is waiting to acquire a resource. A blocked process initiates deadlock detection by sending query messages to all the processes from whom it is waiting to receive a message (these processes are called the dependent set of the process).

If an active process receives a query or reply message, it discards it. When a blocked process \(P_k\) receives a query\((i, j, k)\) message, it takes the following actions:

- If this is the first query message received by \(P_k\) for the deadlock detection initiated by \(P_i\) (called the engaging query), then it propagates the query to all the processes in its dependent set and sets a local variable \(nuk(i)\) to the number of query messages sent.
- If this is not an engaging query, then \(P_k\) returns a reply message to it immediately, provided \(P_k\) has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query.
A local boolean variable \( wait_k(i) \) at process \( P_k \) denotes the fact that it has been continuously blocked since it received the last engaging query from process \( P_i \). When a blocked process \( P_k \) receives a reply \( (i, j, k) \) message, it decrements \( num_k(i) \) provided \( wait_k(i) \) holds. A process sends a reply message in response to an engaging query only after it has received a reply to every query message it sent out for this engaging query; i.e., \( num(i) = 0 \) at it.

An initiator detects a deadlock when it receives reply messages to all the query messages it had sent out.

**THE ALGORITHM.** We now describe the Chandy et al.'s diffusion computation based deadlock detection algorithm in pseudocode for the OR request model [5], [18]:

**Initiate a diffusion computation for a blocked process \( P_i \):**

send query \( (i, i, j) \) to all processes \( P_j \) in the dependent set \( DS_i \) of \( P_i \);

\( num_i(i) := |DS_i|; \ wait_i(i) := true; \)

**When a blocked process \( P_k \) receives a query \( (i, j, k) \):**

if this is the engaging query for process \( P_k \)
then send query \( (i, k, m) \) to all \( P_m \) in its dependent set \( DS_k \);

\( num_k(i) := |DS_k|; \ wait_k(i) := true \)
else if \( wait_k(i) \) then send a reply \( (i, k, j) \) to \( P_j \).

**When a process \( P_k \) receives a reply \( (i, j, k) \)**

if \( wait_k(i) \)
then begin

\( num_k(i) := num_k(i) - 1; \)
if \( num_k(i) = 0 \)
then if \( i = k \) then declare a deadlock
else send reply \( (i, k, m) \) to the process \( P_m \),
which sent the engaging query.

In the above description of the algorithm, we assumed that only one diffusion computation is initiated for a process. In practice, several diffusion computations may be initiated for a process (A diffusion computation is initiated every time the process is blocked). However, note that at any time only one diffusion computation is current for any process. All others are outdated. The current diffusion computation can be distinguished from outdated ones by using sequence numbers (see [5]).

**7.7.4 A Global State Detection Based Algorithm**

There are three deadlock detection algorithms to detect generalized distributed deadlocks using global state detection approach. The algorithm by Bracha and Toueg [2] consists of two phases. In the first phase, \( \text{react} \) (\( \text{write} \)) and in the second phase \( \text{parse} \) (\( \text{read} \)) generalized deadlocks. The first phase terminates when a process \( P_i \) detects a deadlock. The second phase of the distributed deadlock detection algorithm is reduced to detecting generalized deadlocks.

The Kshemkalyani and Pisters [30] algorithm consists of a fan-out sweep in which all messages inwards (messages in which all messages inwards) or all messages outwards (messages in which all messages outwards) are recorded. Both the outward and inward sweeps, the recording of messages is done, and deadlocked. This algorithm allows sweeps can overlap, and can begin before

**SYSTEM MODE**

other node by a message send event or a message send event per Lamport's clock. A node \( i \) is denoted by \( i \).

A node \( i \) is active to a blocked node \( j \), when \( i \) in the WFG. A Reid [2] when \( p_i \) out of its messages to withdraw the messages.

The sending computation event messages are algorithm.

A node \( i \) has the following:

\( wait_i : \) boolean
\( t_i : \) integer \( (= \) in \( (i) : \) set of \( i \): \) set of \( out(i) : \) set of \( p_i : \) integer \( (= \) \( w_i : \) real \( (= \))
DISTRIBUTED DEADLOCK DETECTION

of two phases. In the first phase, the algorithm records a snapshot of a distributed WFG and in the second phase, the algorithm simulates the granting of requests to check for generalized deadlocks. The second phase is nested within the first phase. Therefore, the first phase terminates after the second phase has terminated. The algorithm by Wang et al. [30] also consists of two phases. In the first phase, the algorithm records a snapshot of the distributed WFG. In the second phase, the static WFG recorded in the first phase is reduced to detect any deadlocks. Both the phases occur serially.

The Kshemkalyani-Singhal algorithm [20] has a single phase, which consists of a fan-out sweep of messages outwards from an initiator process and a fan-in sweep of messages inwards to the initiator process. A sweep of a WFG is a traversal of the WFG in which all messages are sent in the direction of the WFG edges (an outward sweep) or all messages are sent against the direction of the WFG edges (an inward sweep). Both the outward and the inward sweeps are done concurrently in the algorithm. In the outward sweep, the algorithm records a snapshot of a distributed WFG. In the inward sweep, the recorded distributed WFG is reduced to determine whether the initiator is deadlocked. This algorithm deals with the complications introduced because the two sweeps can overlap in time at any process, i.e., the reduction of the WFG at a process can begin before all the WFG edges incident at that process have been recorded.

SYSTEM MODEL. The system has \( n \) nodes, with every node connected to every other node by a logical channel. An event in a computation can be an internal event, a message send event, or a message receive event. Events are assigned timestamps as per Lamport’s clock scheme [21]. The timestamp of an event that occurs at time \( t \) on node \( i \) is denoted by \( t_i \).

The computation messages can either be REQUEST, REPLY or CANCEL messages. A node \( i \) sends \( q_i \) REQUESTs to \( q_i \) other nodes when it blocks (goes from an active to a blocked state) on a \( p_i \)-out-of-\( q_i \) request. When node \( i \) blocks on node \( j \) node \( j \) becomes a successor of node \( i \) and node \( i \) becomes a predecessor of node \( j \) in the WFG. A REPLY message denotes the granting of a request. A node \( i \) unblocks when \( p_i \) out of its \( q_i \) requests are granted. When the node unblocks, it sends CANCEL messages to withdraw the remaining \( q_i - p_i \) requests it had sent.

The sending and receiving of REQUEST, REPLY, and CANCEL messages are computation events. The sending and receiving of deadlock detection algorithm messages are algorithm events.

A node \( i \) has the following local variables to record its state:

- \( \text{wait}_i \): boolean \( := \text{false} \); /*records the current status.*/
- \( t_i \): integer \( := 0 \); /*current time.*/
- \( \text{in}(i) \): set of nodes whose requests are outstanding at \( i \)
- \( \text{out}(i) \): set of nodes on which \( i \) is waiting.
- \( p_i \): integer \( := 0 \); /*the number of replies required for unblocking.*/
- \( w_i \): real \( := 1.0 \); /*weight to detect termination of deadlock detection algorithm.*/
REQUEST SEND(i).
/*Executed by node i when it blocks on a p_i-out-of-q_i request.*/
For every node j on which i is blocked do
  out(i) ← out(i) ∪ \{j\};
  send REQUEST(i) to j;
set p_i to the number of replies needed;
wait_i ← true;

REQUEST RECEIVE(j).
/*Executed by node i when it receives a request made by j.*/
in(i) ← in(i) ∪ \{j\};

REPLY SEND(j).
/*Executed by node i when it replies to a request by j.*/
in(i) ← in(i) - \{j\};
send REPLY(i) to j.

REPLY RECEIVE(j).
/*Executed by node i when it receives a reply from j to its request.*/
if valid reply for the current request
  then begin
    out(i) ← out(i) - \{j\};
    p_i ← p_i - 1;
    p_i = 0 →
      \{wait_i ← false;
      \forall k ∈ out(i), send CANCEL(i) to k;
      out(i) ← \{\} \}
  end

CANCEL RECEIVE(j).
/*Executed by node i when it receives a cancel from j.*/
if j ∈ in(i) then in(i) ← in(i) - \{j\}.

When a node init blocks on a P-out-of-Q request, it initiates the deadlock detection algorithm. The algorithm records part of the WFG that is reachable from init (henceforth, referred to as init’s WFG) in a distributed snapshot [4]; such a distributed snapshot includes only those dependency edges and nodes that form init’s WFG. When multiple nodes block concurrently, they may each initiate the deadlock detection algorithm concurrently. Each invocation of the deadlock detection algorithm is treated independently and is identified by the initiator’s identity and initiator’s timestamp when it is blocked. Every node maintains a local snapshot for the latest deadlock detection algorithm initiated by every other node. We will describe only a single instance of the deadlock detection algorithm.

AN INFORMAL record of the WFG is recorded using ECHO messages, using ECHO messages, where the WFG includes its local WFG when it blocks. The WFG is recorded using ECHO messages, and FLOOD messages are used to propagate the WFG in the outward direction, i.e., to record the outward dependencies of the WFG by returning the states of all incomplete nodes and leaf nodes.

ECHO messages are used to request granting of requests. Requests are granted. An edge’s request represents an edge that requests i, any transition request for a WFG to reduce. The node request to reduce request for a WFG cannot be reduced. The WFG is imposed between the incoming edges of the node, the deadlock if it occurs.

In general, the WFG has been recorded when a request arrives and begins reducing the WFG. Since the WFG is recorded between the incoming edges and outward edges, it can be carefully manipulated in the WFG recording hand.

TERMINATION DETECTION [16] detects the termination of a request by reducing the WFG. When the first FLOOD is initiated, is distributed among the WFG further. Since the FLOOD further, it: a FLOOD is received in the WFG further, it: a FLOOD is received leaf node to reduce the weight of the ECHO message. An incoming edge of the node, it
AN INFORMAL DESCRIPTION OF THE ALGORITHM. The distributed WFG is recorded using FLOOD messages in the outward sweep and is examined for deadlocks using ECHO messages in the inward sweep. To detect a deadlock, the initiator \textit{init} records its local state and sends FLOOD messages along its outward dependencies when it blocks. When node \textit{i} receives the first FLOOD message along an existing inward dependency, it records its local state. If node \textit{i} is blocked at this time, it sends out FLOOD messages along its outward dependencies to continue the recording of the WFG in the outward sweep. If node \textit{i} is active at this time, (i.e., it does not have any outward dependencies and is a leaf node in the WFG), then it initiates reduction of the WFG by returning an ECHO message along the incoming dependency even before the states of all incoming dependencies have been recorded in the WFG snapshot at the leaf node.

ECHO messages perform the reduction of the recorded WFG by simulating the granting of requests in the inward sweep. A node \textit{i} in the WFG is reduced if it receives ECHOs along \textit{p}_{\textit{i}} out of its \textit{q}_{\textit{i}} outgoing edges indicating that \textit{p}_{\textit{i}} of its requests can be granted. An edge is reduced if an ECHO is received on the edge indicating that the request it represents can be granted. After a local snapshot has been recorded at node \textit{i}, any transition made by \textit{i} from an idle to an active state is captured in the process of reduction. The nodes that can be reduced do not form a deadlock whereas the nodes that cannot be reduced are deadlocked. The order in which the reduction of the nodes and edges of the WFG is performed does not alter the final result. Node \textit{init} detects the deadlock if it is not reduced when the deadlock detection algorithm terminates.

In general, WFG reduction can begin at a nonleaf node before the recording of the WFG has been completed at that node. This happens when an ECHO message arrives and begins reduction at a nonleaf node before all the FLOODs have arrived and recorded the complete local WFG at that node. Thus, the activities of recording and reducing the WFG snapshot are done concurrently in a single phase and no serialization is imposed between the two activities as is done in [30]. Since a reduction is done on an incompletely recorded WFG at the nodes, the local snapshot at each node has to be carefully manipulated so as to give the effect that WFG reduction is initiated after WFG recording has been completed.

TERMINATION DETECTION. A termination detection technique based on weights [16] detects the termination of the algorithm using SHORT messages (in addition to FLOODs and ECHOs). A weight of 1.0 at the initiator node, when the algorithm is initiated, is distributed among all FLOOD messages sent out by the initiator. When the first FLOOD is received at a nonleaf node, the weight of the received FLOOD is distributed among the FLOODs sent out along outward edges at that node to expand the WFG further. Since any subsequent FLOOD arriving at a nonleaf node does not expand the WFG further, its weight is returned to the initiator through a SHORT message. When a FLOOD is received at a leaf node, its weight is transferred to the ECHO sent by the leaf node to reduce the WFG. When an ECHO that arrives at a node unblocks the node, the weight of the ECHO is distributed among the ECHOs that are sent by that node along the incoming edges in its WFG snapshot. When an ECHO arriving at a node does not unblock the node, its weight is sent directly to the initiator through a SHORT message.
The algorithm maintains the invariant such that the sum of the weights in FLOOD, ECHO, and SHORT messages plus the weight at the initiator (received in SHORT and ECHO messages) is always one. The algorithm terminates when the weight at the initiator becomes 1.0, signifying that all WFG recording and reduction activity has completed.

**THE ALGORITHM.** FLOOD, ECHO, and SHORT control messages use weights (introduced in [16]) for termination detection. The weight $w$ is a real number in the range $[0, 1]$.

A node $i$ stores the local snapshot for snapshots initiated by every other node to detect deadlock in a data structure $LS$, which is an array of records.

$LS$: array $[1..N]$ of record;

A record has several fields to record snapshot related information and is defined below for initiator $init$:

$LS[init].out$: set of integers ($:= \emptyset$); /* nodes on which $i$ is waiting in the snapshot. */

$LS[init].in$: set of integers ($:= \emptyset$); /* nodes waiting on $i$ in the snapshot */

$LS[init].t$: integer ($:= 0$); /* time when $init$ initiated snapshot. */

$LS[init].s$: boolean ($:= false$); /* local blocked state as seen by snapshot. */

$LS[init].p$: integer; /* value of $p_i$ as seen in snapshot. */

The deadlock detection algorithm is defined by the following procedures.

**SNAPSHOT\_INITIATE.**

/* Executed by node $i$ to detect whether it is deadlocked. */

\[
init \leftarrow i; \\
\text{ send } FLOOD(i, i, t_i, 1/|out(i)|) \text{ to each } j \in out(i). /* 1/|out(i)| \text{ is the fraction of weight sent in a FLOOD message. */}
\]

**FLOOD\_RECEIVE($j, init, t.init, w$).**

/* Executed by node $i$ on receiving a FLOOD message from $j$. */

\[
LS[init].t \leftarrow t.init; \\
\text{ send } FLOOD(i, j, t.init, 1/|out(j)|) \text{ to each } k \in in(i). /* Valid FLOOD for a */
\]

/* new snapshot. */

\[
\text{ send } ECHO(j, i, t.init, \text{wait}); \\
\text{ send } ECHO(i, j, t.init, \text{wait}); \\
\text{ send } ECHO(i, i, t.init, \text{wait});
\]

The deadlock detection algorithm is defined by the following procedures.
ights in FLOOD, and in SHORT and SHORT messages use weights and the weight at the

s the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has

s use weights and the weight at the main activity has

cation activity has
SHORT_RECEIVE\(\text{init}, \ t_{\text{init}}, \ w\)

/*Executed by node \(i\) (which is always \(\text{init}\)) on receiving a SHORT. */

/*SHORT for out-dated snapshot. */
\(t_{\text{init}} < t_{\text{block}_i}\) → discard the message.

/*SHORT for uninitiated snapshot. */
\(t_{\text{init}} > t_{\text{block}_i}\) → not possible.

/*SHORT for currently initiated snapshot. */
\(t_{\text{init}} = t_{\text{block}_i} \land LS[\text{init}].s = \text{false}\) → discard.

\(t_{\text{init}} = t_{\text{block}_i} \land LS[\text{init}].s = \text{true}\) →
\(w_i ← w_i + w;\)
\(w_i = 1\) → declare deadlock and abort.

The algorithm has a message complexity of \(4e - 2n + 2l\) and a time complexity of \(2d\) hops, where \(e\) is the number of edges, \(n\) the number of nodes, \(l\) the number of leaf nodes, and \(d\) the diameter of the WFG. This is better than the two-phase algorithms of Bracha and Toueg [2] and Wang et al. [30] and gives the best time complexity of any algorithm that reduces a distributed WFG to detect generalized distributed deadlocks.

7.8 HIERARCHICAL DEADLOCK DETECTION ALGORITHMS

In hierarchical algorithms, sites are (logically) arranged in hierarchical fashion, and a site is responsible for detecting deadlocks involving only its children sites. These algorithms take advantage of access patterns that are localized to a cluster of sites to optimize performance.

7.8.1 The Menasce-Muntz Algorithm

In the hierarchical deadlock detection algorithm of Menasce and Muntz [22], all the controllers are arranged in tree fashion. (A controller manages a resource or is responsible for deadlock detection.) The controllers at the bottom-most level (called leaf controllers) manage resources and others (called nonleaf controllers) are responsible for deadlock detection. A leaf controller maintains a part of the global TWF graph concerned with the allocation of the resources at that leaf controller. A nonleaf controller maintains all TWF graphs spanning its children controllers and is responsible for detecting all deadlocks involving all of its leaf controllers.

Whenever a change occurs in a controller's TWF graph due to a resource allocation, wait, or release, it is propagated to its parent controller. The parent controller makes changes in its TWF graph, searches for cycles, and propagates the changes upward, if necessary. A nonleaf controller can receive up-to-date information concerning the TWF graph of its children continuously (i.e., whenever a change occurs) or periodically.
time complexity of
the number of leaf
phase algorithms of
complexity of any
distributed deadlocks.

In the hierarchical fashion, and
children sites. These
cluster of sites to

Huntz [22], all the
resource or is re-
t level (called leaf
s) are responsible
local TWFG graph
r. A nonleaf con-
control site for each cluster (Fig. 7.2). The central control site
requests from every control site their intercluster transaction status information and
wait-for relations.

As a result, a control site collects status tables from all the sites in its cluster and
applies the one-phase deadlock detection algorithm to detect all deadlocks involving
only intracluster transactions. Then, it sends intercluster transaction status information
and wait-for relations (derived from the information thus collected) to the central control
site. The central site splices the intercluster information it receives, constructs a system
WFG, and searches it for cycles. Thus, a control site detects all deadlocks located in
its cluster, and the central control site detects all intercluster deadlocks.

7.9 PERSPECTIVE

We now discuss issues related to deadlock detection in distributed systems that require
further research.

Theory of Correctness of Algorithms

There is a great dearth of formal methods to prove the correctness of deadlock detection
algorithms for distributed systems. Researchers have often used informal or intuitive

FIGURE 7.2
Hierarchical organization in the Ho-Ramamoorthy algorithm.

7.8.2 The Ho-Ramamoorthy Algorithm

In the hierarchical algorithm of Ho and Ramamoorthy [14], sites are grouped into
several disjoint clusters. Periodically, a site is chosen as a central control site, which
dynamically chooses a control site for each cluster (Fig. 7.2). The central control site
requests from every control site their intercluster transaction status information and
wait-for relations.

As a result, a control site collects status tables from all the sites in its cluster and
applies the one-phase deadlock detection algorithm to detect all deadlocks involving
only intracluster transactions. Then, it sends intercluster transaction status information
and wait-for relations (derived from the information thus collected) to the central control
site. The central site splices the intercluster information it receives, constructs a system
WFG, and searches it for cycles. Thus, a control site detects all deadlocks located in
its cluster, and the central control site detects all intercluster deadlocks.
arguments to show the correctness of their algorithms. However, intuition has proved to be highly unreliable as more than half the algorithms have been found incorrect. A formal proof of the correctness of deadlock detection algorithms becomes nontrivial due to the following factors: (1) TWF graph and deadlock cycles can form in innumerable ways and it is difficult to imagine, let alone exhaustively study, every conceivable situation, (2) deadlock is very sensitive to the timing of requests, and (3) in distributed systems, message delays are unpredictable and there is no global memory. There is a tremendous need for more sophisticated methods to prove the correctness of deadlock detection algorithms.

Performance of the Algorithms

Although a large number of deadlock detection algorithms have been proposed for distributed systems, their performance analysis has not received sufficient attention. Most authors (e.g., Obermarck [24] and Sinha-Natarajan [28]) have evaluated their algorithms on the basis of the number of messages exchanged to detect an existing cycle in the TWF graph. This performance criterion is deceptive because deadlock detection algorithms also exchange messages during normal conditions (when there is no deadlock). The number of messages exchanged may not be the true indicator of the communication overhead because some algorithms (e.g., [12, 17, 24]) may exchange large messages as opposed to other algorithms (e.g., [5], [22]) which exchange small messages. Therefore, we require a different criterion for computing the communication overhead, which should take into account the number as well as the size of messages exchanged, not only in deadlocked conditions but also in normal conditions.

The persistence of deadlocks results in the wasteful utilization of resources and increased response delay to user requests. Therefore, an important performance measure of deadlock detection algorithms is the average time a deadlock persists; we term this deadlock persistence time. There is often a tradeoff between message traffic, and deadlock persistence time. For example, the on-line deadlock detection algorithm of Isloor and Marsland [17] detects a deadlock at the earliest instant but it has high message traffic. On the other hand, the algorithm of Obermarck [24] has less message traffic, but the deadlock persistence time in it is proportional to the size of the cycle.

Besides communication overhead and deadlock persistence time, any evaluation of deadlock detection algorithms should also consider measures such as storage overhead to store deadlock detection information, processing overhead to search for cycles, and additional processing required to optimally resolve a deadlock. The factors that influence these measures are the techniques used for deadlock detection, the data access behavior of processes, the request-release pattern of processes, resource holding time, etc. How these factors influence the performance of different deadlock detection algorithms, and how the performance of different deadlock detection algorithms compare with each other, are not well understood and are still open issues. A complete performance study of deadlock detection algorithms calls for the development of performance models, the determination of the aforementioned performance metrics using analytic or simulation techniques, and a comparison of the performance of existing deadlock detection algorithms.

Deadlock Resolution

The persistence of deadlocks results in the wasteful utilization of resources and increased response delay to user requests. Therefore, an important performance measure of deadlock detection algorithms is the average time a deadlock persists; we term this deadlock persistence time. There is often a tradeoff between message traffic, and deadlock persistence time. For example, the on-line deadlock detection algorithm of Isloor and Marsland [17] detects a deadlock at the earliest instant but it has high message traffic. On the other hand, the algorithm of Obermarck [24] has less message traffic, but the deadlock persistence time in it is proportional to the size of the cycle.

Besides communication overhead and deadlock persistence time, any evaluation of deadlock detection algorithms should also consider measures such as storage overhead to store deadlock detection information, processing overhead to search for cycles, and additional processing required to optimally resolve a deadlock. The factors that influence these measures are the techniques used for deadlock detection, the data access behavior of processes, the request-release pattern of processes, resource holding time, etc. How these factors influence the performance of different deadlock detection algorithms, and how the performance of different deadlock detection algorithms compare with each other, are not well understood and are still open issues. A complete performance study of deadlock detection algorithms calls for the development of performance models, the determination of the aforementioned performance metrics using analytic or simulation techniques, and a comparison of the performance of existing deadlock detection algorithms.
The persistence of a deadlock has two major undesirable effects: first, the resources held by deadlocked processes are not available to any other process, and second, the deadlock persistence time is added to the response time of each process involved in the deadlock. Therefore, the problem of promptly and efficiently resolving a detected deadlock is as important as the problem of deadlock detection itself. Unfortunately, many deadlock detection algorithms for distributed systems do not address the problem of deadlock resolution.

A deadlock is resolved by aborting at least one process involved in the deadlock and granting the released resources to other processes involved in the deadlock. The efficient resolution of a deadlock requires knowledge of all the processes involved in the deadlock and all resources held by these processes. When a deadlock is detected, the quickness of its resolution depends in great measure on how much information about it is available, which in turn depends on how much information is circulated during the deadlock detection phase. In many distributed deadlock detection algorithms, deadlock resolution is complicated by at least one of the following problems:

- A process that detects a deadlock does not know all the processes (and resources held by them) involved in the deadlock, e.g., Chandy et al. [5] and Menasce-Muntz [22].
- Two or more processes may independently detect the same deadlock, e.g., Chandy et al. [5] and Goldman [10]. If every process that detects a deadlock resolves it, then deadlock resolution will be inefficient as several processes will be aborted to resolve a deadlock (different processes may choose to abort different processes). Therefore, we require some post-detection processing to select a process that is responsible for resolving the deadlock.

After the idea of assigning unique priorities to transactions/processes was introduced by Obermarck [24], it has been successfully used in attacking the above problems in the following manner:

- Each deadlock is detected only by the highest priority process in the deadlock (deadlock detections initiated by all other deadlocked processes are suppressed).
- When the highest priority process detects a deadlock, it knows the lowest priority processes in the deadlock cycle, which can be aborted to resolve the deadlock.

The Sinha-Natarajan algorithm [28] is an excellent example of the above techniques. It should be noted that the lowest priority process that is selected for abortion (called the victim) may not necessarily result in an optimal resolution of the deadlock in the classical sense.

Even after the above two problems are solved, the resolution of a deadlock involves the following nontrivial steps:
1. The victim must be aborted, all the resources held by it must be released, the state of all the released resources must be restored to their previous states, and the released resources must be granted to deadlocked processes.

2. All the deadlock detection information concerning the victim must be cleaned at all the sites.

The execution of Step 1 is complicated in environments where a process can simultaneously wait for multiple resources, because a deadlock can be caused by the allocation of a released resource to another process. The execution of Step 2 is even more critical because if the information about the victim is not cleaned quickly, it may be counted in several other (false) cycles causing detection of false deadlocks. As has been pointed out in Choudhary et al. [7], the lack of proper cleaning of probe messages in Sinha-Natarajan [28] causes the detection of false deadlocks. To be safe, during the execution of Step 1 and Step 2, the deadlock detection activity (at least in those potential deadlocks that include the victim) must be halted to avoid the detection of false deadlocks. In Sugihara et al. [29], a control token is used to serialize the resolution of global deadlocks. This simplifies the elimination of side effects of deadlock resolution on the deadlock detection activity.

False Deadlocks. In environments where a process can simultaneously wait for multiple resources, deadlock resolution becomes nontrivial. This is because an edge may be shared by two or more cycles and the deletion of that edge will break all those deadlocks. However, since the search for each cycle is carried out independently, deadlock detection initiated for some cycles may not be aware of the deleted edge, resulting in the detection of false or phantom deadlocks [27].

Deadlock detection involves detecting a static condition because once a deadlock cycle is formed, it persists until it is detected and broken. On the other hand, deadlock resolution is a dynamic activity because it changes the WFG by deleting its edges and nodes. There are two forces working in opposite directions: the wait for resources adds edges/nodes to the WFG, while deadlock resolution removes edges/nodes from the WFG. Therefore, if deadlock resolution is not carefully incorporated into deadlock detection, false deadlocks are likely to be detected.

7.10 SUMMARY

Of the three approaches to handle deadlocks, deadlock detection is the most promising for distributed systems. The detection of deadlocks requires performing two tasks: first, maintaining (or constructing whenever needed) a WFG; second, searching the WFG for cycles. Depending upon the way the WFG is maintained and the way a control to carry out the search for cycles is structured, deadlock detection algorithms are classified into three categories: centralized, distributed, and hierarchical.

In centralized deadlock detection algorithms, the control site has the responsibility of constructing the global state graph and searching it for cycles. Centralized deadlock detection algorithms are conceptually simple and easy to implement. However, centralized deadlock detection algorithms have a single point of failure, communication links near the control site with deadlock detection

In distributed deadlock detection, the global state graph is maintained at all the sites. Distributed algorithms are designed to detect deadlocks in environments even though all the processes are running concurrently.

Hierarchical deadlock detection algorithms are designed to detect deadlocks in systems consisting of multiple processes. These systems are divided into sub-systems, and each sub-system has its own control site. These control sites communicate with each other to detect cycles which may span across sub-systems.

Of the three deadlock detection approaches, detection is the most widely used because it is conceptually simple and easy to implement. Distributed deadlock detection algorithms are conceptually simple and easy to implement. However, centralized deadlock detection algorithms have a single point of failure, communication links near the control site with deadlock detection.

In distributed deadlock detection, the global state graph is maintained at all the sites. Distributed algorithms are designed to detect deadlocks in environments even though all the processes are running concurrently.

Hierarchical deadlock detection algorithms are designed to detect deadlocks in systems consisting of multiple processes. These systems are divided into sub-systems, and each sub-system has its own control site. These control sites communicate with each other to detect cycles which may span across sub-systems.

Of the three deadlock detection approaches, detection is the most widely used because it is conceptually simple and easy to implement. Distributed deadlock detection algorithms are conceptually simple and easy to implement. However, centralized deadlock detection algorithms have a single point of failure, communication links near the control site with deadlock detection.
DISTRIBUTED DEADLOCK DETECTION

When a process can accumulate wait-for dependency messages, the released wait-for dependency message can cause another process to be released, the state of which must be cleaned at the control site. This can lead to a process becoming deadlocked if Step 2 is even slightly delayed, and it may also lead to the accumulation of readlocks. As has been discussed in [7], the release of probe messages from blocked processes may need to be safe, during a period in which the WFG is not being updated. As a result, the resolution of deadlocks may involve the deadlock resolution algorithms discussed in [13].

Deadlocks can be detected by waiting for multi-process wait-for dependencies. However, an edge may be deleted if it is not needed, and in the worst case, all the deadlocks involving an edge are detected independently, deadlock can occur when multiple processes wait for resources which are not available. Consequently, deleting its edges from the WFG may result in a cycle, resulting in the detection of a new deadlock.

Hierarchical deadlock detection algorithms fall between centralized and distributed deadlock detection algorithms and exploit access patterns local to a cluster of sites to detect deadlocks efficiently. These algorithms do not have a single point of failure (as in centralized algorithms) and a site is not bogged down by the deadlock detection activities which it is not much concerned with (as in some distributed algorithms).

Distributed deadlock detection algorithms can be divided into four classes: path-pushing, edge-chasing, diffusion computation, and global state detection. In path-pushing algorithms, wait-for dependency information of the global WFG is disseminated in the form of paths (i.e., a sequence of wait-for dependency edges). In edge-chasing algorithms, special messages called probes are circulated along the edges of the WFG to detect a cycle. When a blocked process receives a probe, it propagates the probe along its outgoing edges in the WFG. A process declares a deadlock when it receives a probe initiated by it. Diffusion computation type algorithms make use of echo algorithms to detect deadlocks. Deadlock detection messages are successively propagated (i.e., “diffused”) through the edges of the WFG. Global state detection based algorithms detect deadlocks by taking a snapshot of the system and by examining it for the condition of a deadlock.

7.11 FURTHER READING

Two up-to-date survey articles on distributed deadlock detection can be found in papers by Knapp [18] and Singhal [27]. Badal [1] discusses a distributed deadlock detection algorithm that optimizes performance by first detecting a deadlock using a simple algorithm and then successively using more complex algorithms if a deadlock remains undetected. Other path-pushing distributed deadlock detection algorithms can be found in papers by Gligor and Shattuck [9], Goldman [10], and Menasce and Muntz [22]. Other edge-chasing distributed deadlock detection algorithms can be found in papers by Choudary et al. [7], Roesler and Burkhard [25], and Kshemkalyani and Singhal [19]. Herman and Chandy [13] discuss detection of deadlocks in the AND/OR request model.
Sanders and Heuberger [26] synthesize an edge-chasing distributed deadlock detection algorithm from the first principles. Distributed deadlock detection in CSP-like environments are discussed in Elmagarmid et al. [8] and Huang [15]. Two rigorous correctness proof of distributed deadlock detection algorithms appear in Roesler and Burkhard [25] and Kshemkalyani and Singhal [19].

PROBLEMS

7.1. Distributed deadlock detection algorithms normally have substantial message overhead, even when there is no deadlock. Instead of using a deadlock detection algorithm, we can handle deadlocks in distributed systems simply by using "timeouts" where a process that has waited for a specified period for a resource declares that it is deadlocked (and aborts to resolve the deadlock). What are the risks in using this method?

7.2. Discuss the impact of a message loss on the various deadlock detection algorithms discussed in this chapter.

7.3. Suppose all the processes in the system are assigned priorities that can be used to totally order the processes. Modify Chandy et al.’s algorithm in Sec. 7.7.2 so that when a process detects a deadlock, it also knows the lowest priority deadlocked process. Give an example to show that in the AND request model, false deadlocks can occur due to deadlock resolution in distributed systems [27]. Can something be done about it or are they bound to happen?

7.4. Consider the following scheme to reduce message traffic in distributed deadlock detection [28]: Transactions are assigned unique priorities, and an antagonistic conflict occurs when a transaction waits for a data object that is locked by a lower priority transaction. A deadlock detection is initiated only when an antagonistic conflict occurs. When a waiting transaction receives a probe that is initiated by a lower priority transaction, the probe is discarded.

(a) Determine the number of messages exchanged to detect a deadlock in the “best” case.

(b) Determine the number of messages exchanged to detect a deadlock in the “worst” case.

(c) Determine the number of messages exchanged to detect a deadlock in an “average” case.

(d) Determine the saving (as a percentage) in the average number of messages exchanged under this message traffic reduction scheme, as compared to when no such scheme is used.

REFERENCES


REFERENCES


11 DISTRIBUTED SCHEDULING

11.1 INTRODUCTION

Distributed systems offer a tremendous processing capacity. However, in order to realize this tremendous computing capacity, and to take full advantage of it, good resource allocation schemes are needed. A distributed scheduler is a resource management component of a distributed operating system that focuses on judiciously and transparently redistributing the load of the system among the computers such that overall performance of the system is maximized. Because wide-area networks have high communication delays, distributed scheduling is more suitable for distributed systems based on local area networks.

In this chapter, we discuss several key issues in load distributing, including the motivation for load distributing, tradeoffs between load balancing and load sharing and between preemptive and nonpreemptive task transfers, and stability. In addition, we describe several load distributing algorithms and compare their performance. Surveys of load distributing policies and task migration mechanisms that have been implemented are also presented. This chapter is based on [32].

11.2 MOTIVATION

A locally distributed system consists of a collection of autonomous computers, connected by a local area communication network (Fig. 11.1). Users submit tasks at their
host computers for processing. The need for load distributing arises in such environments because, due to the random arrival of tasks and their random CPU service time requirements, there is a good possibility that several computers are heavily loaded (hence suffering from performance degradation), while others are idle or lightly loaded.

Clearly, if the workload at some computers is typically heavier than that at others, or if some processors execute tasks at a slower rate than others, this situation is likely to occur often. The usefulness of load distributing is not as obvious in systems in which all processors are equally powerful and, over the long term, have equally heavy workloads. Livny and Melman [24] have shown that even in such homogeneous distributed systems, statistical fluctuations in the arrival of tasks and task service time requirements at computers lead to the high probability that at least one computer is idle while a task is waiting for service elsewhere. Their analysis, presented next, models a computer in a distributed system by an $M/M/1$ server.

Consider a system of $N$ identical and independent $M/M/1$ servers [16]. By identical we mean that all servers have the same task arrival and service rates. Let $\rho$ be the utilization of each server. Then $P_o = 1 - \rho$ is the probability that a server is idle. Let $P$ be the probability that the system is in a state in which at least one task is waiting for service and at least one server is idle. Then $P$ is given by the expression [24]

$$P = \sum_{i=1}^{N} P \left( \sum_{i} \frac{Q_i}{N} \right)$$

where $Q_i$ is the probability that a given task is waiting for service when $i$ servers are busy. Thus,

$$H_{N-i} = \left( \text{probability that all } (N-i) \text{ systems have more than one task waiting} \right)$$

Therefore,

$$P = \sum_{i=1}^{N} P = \sum_{i=1}^{N} \left( N \cdot \left( \frac{1}{N} \right) \cdot \left( \frac{1}{N} \right) \right)$$

Figure 11.2 plots the number of servers as a function of the utilization $\rho$. The value of $P$ is high if all servers are heavily loaded, and low if some computers (senders) are lightly loaded. At high system utilizations, the probability of load distribution is high; at lower utilizations, this probability is lower.

Therefore, even if the load is potentially improved by distributing tasks among computers (senders) to different sets of hosts, the following two questions need to be answered: what is the proper characterization of load as a performance metric? And how much load should be redistributed to avoid high average response times? Also, it is crucial that this be done with minimal overhead. These are the questions that we address in the next section.
DISTRIBUTED SCHEDULING

\[ P = \sum_{i=1}^{N} \left( \begin{array}{c} N \\ i \end{array} \right) Q_i H_{N-i} \]  

(11.1)

where \( Q_i \) is the probability that a given set of \( i \) servers are idle and \( H_{N-i} \) is the probability that a given set of \((N - i)\) servers are not idle and at one or more of them a task is waiting for service. Clearly, from the independence assumption,

\[ Q_i = P_0^i \]  

(11.2)

\[ H_{N-i} = \{\text{probability that } (N - i) \text{ systems have at least one task}\} - \{\text{probability that all } (N - i) \text{ systems have exactly one task}\}. \]

Therefore,

\[ P = \sum_{i=1}^{N} \left( \begin{array}{c} N \\ i \end{array} \right) P_0^i \left\{ (1 - P_0)^{N-i} - [(1 - P_0)P_o]^{N-i} \right\} \]

\[ = \sum_{i=1}^{N} \left( \begin{array}{c} N \\ i \end{array} \right) P_0^i(1 - P_0)^{N-i} - \sum_{i=1}^{N} \left( \begin{array}{c} N \\ i \end{array} \right) P_o^N (1 - P_0)^{N-i} \]

\[ = \left\{ 1 - (1 - P_0)^N \right\} - \left\{ P_o^N \left[ (2 - P_0)^N - (1 - P_0)^N \right] \right\} \]

\[ = 1 - (1 - P_0)^N - P_o^N (2 - P_0)^N \]  

(11.4)

Figure 11.2 plots the values of \( P \) for various values of server utilizations \( \rho \) and the number of servers \( N \). For moderate system utilization (where \( \rho = 0.5 \) to 0.8), the value of \( P \) is high, indicating a good potential for performance improvement through load distribution. At high system utilizations, the value of \( P \) is low as most servers are likely to be busy, which indicates lower potential for load distribution. Similarly, at low system utilizations, the value of \( P \) is low as most servers are likely to be idle, which indicates lower potential for load distribution. Another important observation is that, as the number of servers in the system increase, \( P \) remains high even at high system utilizations.

Therefore, even in a homogeneous distributed system, system performance can potentially be improved by appropriately transferring the load from heavily loaded computers (senders) to idle or lightly loaded computers (receivers). This raises the following two questions. First, what is meant by performance? One widely used performance metric is the average response time of tasks. The response time of a task is the length of the time interval between its origination and completion. Minimizing the average response time is often the goal of load distributing. Second, what constitutes a proper characterization of load at a node? Defining a proper load index is very important as load distributing decisions are based on the load measured at one or more nodes. Also, it is crucial that the mechanism used to measure load is efficient and imposes minimal overhead. These issues are discussed next.
11.3 ISSUES IN LOAD DISTRIBUTING

We now discuss several central issues in load distributing that will help the reader understand its intricacies. Note here that the terms computer, machine, host, workstation, and node are used interchangeably, depending upon the context.

11.3.1 Load

Zhou [41] showed that resource queue lengths and particularly the CPU queue length are good indicators of load because they correlate well with the task response time. Moreover, measuring the CPU queue length is fairly simple and carries little overhead. If a task transfer involves significant delays, however, simply using the current CPU queue length as a load indicator can result in a node accepting tasks while other tasks it accepted earlier are still in transit. As a result, when all the tasks that the node has accepted have arrived, the node can become overloaded and require further task transfers to reduce its load. This undesirable situation can be prevented by artificially incrementing the CPU queue length at a node whenever the node accepts a remote task. To avoid anomalies when task transfers fail, a timeout (set at the time of acceptance) can be employed. After the timeout, if the task has not yet arrived, the CPU queue length is decremented.

While the CPU queue length has been extensively used in previous studies as a load indicator, it has been reported that little correlation exists between CPU queue length and processor utilization [35], particularly in an interactive environment. Hence, the designers of V-System used CPU utilization as an indicator of the load at a site. This approach requires a background process that monitors CPU utilization continuously and imposes more overhead, compared to simply finding the queue length at a node (see Sec. 11.10.1).

11.3.2 Classification

The basic functionality loaded computers can be broadly classified into algorithms based on the task's nature and the computer's environment. At least two algorithms can be distinguished: hard-wired algorithms and algorithms that adapt to the environment (at nodes), at least in part. However, the distinction is not always clear-cut, and the algorithms can be hard to distinguish. For example, some algorithms [3, 15] adapt their behavior to suit the environment, while others do not. The latter algorithms are likely to be less sensitive to changes in the environment.

11.3.3 Load Balancing

Load distributing algorithms, based on load-balancing algorithms, can be broadly classified into algorithms that adapt their behavior to suit the environment. These algorithms can be further divided into algorithms that adapt their behavior to suit the environment on an interactive system and algorithms that adapt to the environment of a large, interactive system. The former algorithms are likely to be more efficient, while the latter algorithms are likely to be more flexible. However, the distinction is not always clear-cut, and the algorithms can be hard to distinguish.

11.3.4 Preemptive Task Transfer

Preemptive task transfer is an expensive type of task transfer. However, some algorithms, such as the V-System, use preemptive task transfer to avoid the problems associated with non-preemptive task transfer. These algorithms are likely to be more efficient, while the latter algorithms are likely to be more flexible. However, the distinction is not always clear-cut, and the algorithms can be hard to distinguish.
11.3.2 Classification of Load Distributing Algorithms

The basic function of a load distributing algorithm is to transfer load (tasks) from heavily loaded computers to idle or lightly loaded computers. Load distributing algorithms can be broadly characterized as static, dynamic, or adaptive. Dynamic load distributing algorithms [3, 10, 11, 18, 20, 24, 31, 34, 40] use system state information (the loads at nodes), at least in part, to make load distributing decisions, while static algorithms make no use of such information. In static load distributing algorithms, decisions are hard-wired in the algorithm using a priori knowledge of the system. Dynamic load distributing algorithms have the potential to outperform static load distributing algorithms because they are able to exploit short term fluctuations in the system state to improve performance. However, dynamic load distributing algorithms entail overhead in the collection, storage, and analysis of system state information. Adaptive load distributing algorithms [20, 31] are a special class of dynamic load distributing algorithms in that they adapt their activities by dynamically changing the parameters of the algorithm to suit the changing system state. For example, a dynamic algorithm may continue to collect the system state irrespective of the system load. An adaptive algorithm, on the other hand, may discontinue the collection of the system state if the overall system load is high to avoid imposing additional overhead on the system. At such loads, all nodes are likely to be busy and attempts to find receivers are unlikely to be successful.

11.3.3 Load Balancing vs. Load Sharing

Load distributing algorithms can further be classified as load balancing or load sharing algorithms, based on their load distributing principle. Both types of algorithms strive to reduce the likelihood of an unshared state (a state in which one computer lies idle while at the same time tasks contend for service at another computer [21]) by transferring tasks to lightly loaded nodes. Load balancing algorithms [7, 20, 24], however, go a step further by attempting to equalize loads at all computers. Because a load balancing algorithm transfers tasks at a higher rate than a load sharing algorithm, the higher overhead incurred by the load balancing algorithm may outweigh this potential performance improvement.

Task transfers are not instantaneous because of communication delays and delays that occur during the collection of task state. Delays in transferring a task increase the duration of an unshared state as an idle computer must wait for the arrival of the transferred task. To avoid lengthy unshared states, anticipatory task transfers from overloaded computers to computers that are likely to become idle can be used. Anticipatory transfers increase the task transfer rate of a load sharing algorithm, making it less distinguishable from load balancing algorithms. In this sense, load balancing can be considered a special case of load sharing, performing a particular level of anticipatory task transfers.

11.3.4 Preemptive vs. Nonpreemptive Transfers

Preemptive task transfers involve the transfer of a task that is partially executed. This transfer is an expensive operation as the collection of a task's state (which can be quite
large and complex) can be difficult. Typically, a task state consists of a virtual memory image, a process control block, unread I/O buffers and messages, file pointers, timers that have been set, etc. Nonpreemptive task transfers, on the other hand, involve the transfer of tasks that have not begun execution and hence do not require the transfer of the task's state. In both types of transfers, information about the environment in which the task will execute must be transferred to the receiving node. This information can include the user's current working directory, the privileges inherited by the task, etc. Nonpreemptive task transfers are also referred to as task placements.

11.4 COMPONENTS OF A LOAD DISTRIBUTING ALGORITHM

Typically, a load distributing algorithm has four components: (1) a transfer policy that determines whether a node is in a suitable state to participate in a task transfer, (2) a selection policy that determines which task should be transferred, (3) a location policy that determines to which node a task selected for transfer should be sent, and (4) an information policy which is responsible for triggering the collection of system state information. A transfer policy typically requires information on the local node's state to make decisions. A location policy, on the other hand, is likely to require information on the states of remote nodes to make decisions.

11.4.1 Transfer Policy

A large number of the transfer policies that have been proposed are threshold policies [10, 11, 24, 31]. Thresholds are expressed in units of load. When a new task originates at a node, and the load at that node exceeds a threshold $T$, the transfer policy decides that the node is a sender. If the load at a node falls below $T$, the transfer policy decides that the node can be a receiver for a remote task.

An alternative transfer policy initiates task transfers whenever an imbalance in load among nodes is detected because of the actions of the information policy.

11.4.2 Selection Policy

A selection policy selects a task for transfer, once the transfer policy decides that the node is a sender. Should the selection policy fail to find a suitable task to transfer, the node is no longer considered a sender until the transfer policy decides that the node is a sender again.

The simplest approach is to select newly originated tasks that have caused the node to become a sender by increasing the load at the node beyond the threshold [11]. Such tasks are relatively cheap to transfer, as the transfer is nonpreemptive.

A basic criterion that a task selected for transfer should satisfy is that the overhead incurred in the transfer of the task should be compensated for by the reduction in the response time realized by the task. In general, long-lived tasks satisfy this criterion [4]. Also, a task can be selected for remote execution if the estimated average execution time for that type of task is greater than some execution time threshold [36].

Bryant an
Bryant and Finkel [3] propose another approach based on the reduction in response time that can be obtained for a task by transferring it elsewhere. In this method, a task is selected for transfer only if its response time will be improved upon transfer. (See [3] for details on how to estimate response time.)

There are other factors to consider in the selection of a task. First, the overhead incurred by the transfer should be minimal. For example, a task of small size carries less overhead. Second, the number of location-dependent system calls made by the selected task should be minimal. Location-dependent calls must be executed at the node where the task originated because they use resources such as windows, or the mouse, that only exist at the node [8, 19].

### 11.4.3 Location Policy

The responsibility of a location policy is to find suitable nodes (senders or receivers) to share load. A widely used method for finding a suitable node is through polling. In polling, a node polls another node to find out whether it is a suitable node for load sharing [3, 10, 11, 24, 31]. Nodes can be polled either serially or in parallel (e.g., multicast). A node can be selected for polling either randomly [3, 10, 11], based on the information collected during the previous polls [24, 31], or on a nearest-neighbor basis. An alternative to polling is to broadcast a query to find out if any node is available for load sharing.

### 11.4.4 Information Policy

The information policy is responsible for deciding when information about the states of other nodes in the system should be collected, where it should be collected from, and what information should be collected. Most information policies are one of the following three types:

**Demand-driven.** In this class of policy, a node collects the state of other nodes only when it becomes either a sender or a receiver (decided by the transfer and selection policies at the node), making it a suitable candidate to initiate load sharing. Note that a demand-driven information policy is inherently a dynamic policy, as its actions depend on the system state. Demand-driven policies can be sender-initiated, receiver-initiated, or symmetrically initiated. In sender-initiated policies, senders look for receivers to transfer their load. In receiver-initiated policies, receivers solicit load from senders. A symmetrically initiated policy is a combination of both, where load sharing actions are triggered by the demand for extra processing power or extra work.

**Periodic.** In this class of policy, nodes exchange load information periodically [14, 40]. Based on the information collected, the transfer policy at a node may decide to transfer jobs. Periodic information policies do not adapt their activity to the system state. For example, the benefits due to load distributing are minimal at high system loads because most of the nodes in the system are busy. Nevertheless, overheads due to periodic information collection continue to increase the system load and thus worsen the situation.
State-change-driven. In this class of policy, nodes disseminate state information whenever their state changes by a certain degree [24]. A state-change-driven policy differs from a demand-driven policy in that it disseminates information about the state of a node, rather than collecting information about other nodes. Under centralized state-change-driven policies, nodes send state information to a centralized collection point. Under decentralized state-change-driven policies, nodes send information to peers [32].

11.5 STABILITY
We now describe two views of stability.

11.5.1 The Queuing-Theoretic Perspective
When the long term arrival rate of work to a system is greater than the rate at which the system can perform work, the CPU queues grow without bound. Such a system is termed unstable. For example, consider a load distributing algorithm performing excessive message exchanges to collect state information. The sum of the load due to the external work arriving and the load due to the overhead imposed by the algorithm can become higher than the service capacity of the system, causing system instability.

Alternatively, an algorithm can be stable but may still cause a system to perform worse than when it is not using the algorithm. Hence, a more restrictive criterion for evaluating algorithms is desirable, and we use the effectiveness of an algorithm as the evaluating criterion. A load distributing algorithm is said to be effective under a given set of conditions if it improves the performance relative to that of a system not using load distributing. Note that while an effective algorithm cannot be unstable, a stable algorithm can be ineffective.

11.5.2 The Algorithmic Perspective
If an algorithm can perform fruitless actions indefinitely with finite probability, the algorithm is said to be unstable [3]. For example, consider processor thrashing. The transfer of a task to a receiver may increase the receiver’s queue length to the point of overload, necessitating the transfer of that task to yet another node. This process may repeat indefinitely [3]. In this case, a task is moved from one node to another in search of a lightly loaded node without ever receiving service. Discussions on various types of algorithmic instability are beyond the scope of this book and can be found in [6].

11.6 LOAD DISTRIBUTING ALGORITHMS
We now describe some load distributing algorithms that have appeared in the literature and discuss their performance.
11.6.1 Sender-Initiated Algorithms

In sender-initiated algorithms, load distributing activity is initiated by an overloaded node (sender) that attempts to send a task to an underloaded node (receiver). This section covers three simple yet effective sender-initiated algorithms studied by Eager, Lazowska, and Zohorjan [11].

Transfer policy. All three algorithms use the same transfer policy, a threshold policy based on CPU queue length. A node is identified as a sender if a new task originating at the node makes the queue length exceed a threshold $T$. A node identifies itself as a suitable receiver for a remote task if accepting the task will not cause the node’s queue length to exceed $T$.

Selection policy. These sender-initiated algorithms consider only newly arrived tasks for transfer.

Location policy. These algorithms differ only in their location policy:

Random. Random is a simple dynamic location policy that uses no remote state information. A task is simply transferred to a node selected at random, with no information exchange between the nodes to aid in decision making. A problem with this approach is that useless task transfers can occur when a task is transferred to a node that is already heavily loaded (i.e., its queue length is above the threshold). An issue raised with this policy concerns the question of how a node should treat a transferred task. If it is treated as a new arrival, the transferred task can again be transferred to another node if the local queue length is above the threshold. Eager et al. [11] have shown that if such is the case, then irrespective of the average load of the system, the system will eventually enter a state in which the nodes are spending all their time transferring tasks and not executing them. A simple solution to this problem is to limit the number of times a task can be transferred. A sender-initiated algorithm using the random location policy provides substantial performance improvement over no load sharing at all [11].

Threshold. The problem of useless task transfers under random location policy can be avoided by polling a node (selected at random) to determine whether it is a receiver (see Fig. 11.3). If so, the task is transferred to the selected node, which must execute the task regardless of its state when the task actually arrives. Otherwise, another node is selected at random and polled. The number of polls is limited by a parameter called PollLimit to keep the overhead low. Note that while nodes are randomly selected, a sender node will not poll any node more than once during one searching session of PollLimit polls. If no suitable receiver node is found within the PollLimit polls, then the node at which the task originated must execute the task. By avoiding useless task transfers, the threshold policy provides substantial performance improvement over the random location policy [11].

Shortest. The two previous approaches make no effort to choose the best receiver for a task. Under the shortest location policy, a number of nodes (= PollLimit) are
selected at random and are polled to determine their queue length [11]. The node with the shortest queue length is selected as the destination for task transfer unless its queue length is \( T \). The destination node will execute the task regardless of its queue length at the time of arrival of the transferred task. The performance improvement obtained by using the shortest location policy over the threshold policy was found to be marginal [11], indicating that using more detailed state information does not necessarily result in significant improvement in system performance.

**Information policy.** When either the shortest or the threshold location policy is used, polling activity commences when the transfer policy identifies a node as the sender of a task. Hence, the information policy can be considered to be of the demand-driven type.

**Stability.** These three approaches for location policy used in sender-initiated algorithms cause system instability at high system loads, where no node is likely to be lightly loaded, and hence the probability that a sender will succeed in finding a receiver node is very low. However, the polling activity in sender-initiated algorithms increases as the rate at which work arrives in the system increases, eventually reaching a point where the cost of load sharing is greater than the benefit. At this point, most of the available CPU cycles are wasted in unsuccessful polls and in responding to these polls. When the load due to work arriving and due to the load sharing activity exceeds the system's serving capacity, it is not effective at high system state.

11.6.2 Receiver-initiated

In receiver-initiated algorithms, a loaded node (receiver) will execute a task from a node (sender) if it is to be a sender if its CPU queue length, its length falls below the threshold. In this section, we algorithm proposed

**Transfer policy.** To a CPU queue length, its length falls below the threshold. In this section, we algorithm proposed

**Selection policy.** This policy is achieved under the selection policy

**Location policy.** In this section, we propose an algorithm that transfers a task from a node to another node based on the length of the CPU queue at each node. When a task arrives at a node, it is placed in a queue, and the node checks the length of the CPU queue at each node to determine if it has enough capacity to execute the task. If the node has enough capacity, it transfers the task to a destination node, which is selected based on the location policy. The destination node will execute the task regardless of its queue length at the time of arrival of the transferred task. The performance improvement obtained by using the shortest location policy over the threshold policy was found to be marginal [11], indicating that using more detailed state information does not necessarily result in significant improvement in system performance.
serving capacity, instability occurs. Thus, the actions of sender-initiated algorithms are not effective at high system loads and cause system instability by failing to adapt to the system state.

### 11.6.2 Receiver-Initiated Algorithms

In receiver-initiated algorithms, the load distributing activity is initiated from an underloaded node (receiver) that is trying to obtain a task from an overloaded node (sender). In this section, we describe the policies of an algorithm [31] that is a variant of the algorithm proposed in [10] (see Fig. 11.4).

**Transfer policy.** Transfer policy is a threshold policy where the decision is based on CPU queue length. The transfer policy is triggered when a task departs. If the local queue length falls below the threshold \( T \), the node is identified as a receiver for obtaining a task from a node (sender) to be determined by the location policy. A node is identified to be a sender if its queue length exceeds the threshold \( T \).

**Selection policy.** This algorithm can make use of any of the approaches discussed under the selection policy in Sec. 11.4.2.

**Location policy.** In this policy, a node selected at random is polled to determine if transferring a task from it would place its queue length below the threshold level. If

---

![Figure 11.4](image.png)

**FIGURE 11.4**

Receiver-initiated load sharing.
not, the polled node transfers a task. Otherwise, another node is selected at random and
the above procedure is repeated until a node that can transfer a task (i.e., a sender) is
found, or a static PollLimit number of tries have failed to find a sender. If all polls fail
to find a sender, the node waits until another task completes or until a predetermined
period is over before initiating the search for a sender, provided the node is still a
receiver. Note that if the search does not start after a predetermined period, the extra
processing power available at a receiver is completely lost to the system until another
task completes, which may not occur soon.

Information policy. The information policy is demand-driven because the polling ac­
tivity starts only after a node becomes a receiver.

Stability. Receiver-initiated algorithms do not cause system instability for the following
reason. At high system loads there is a high probability that a receiver will find a suitable
sender to share the load within a few polls. This results in the effective usage of polls
from receivers and very little wastage of CPU cycles at high system loads. At low
system loads, there are few senders but more receiver-initiated polls. These polls do
not cause system instability as spare CPU cycles are available at low system loads.

A drawback. Under the most widely used CPU scheduling disciplines (such as round­
robin and its variants), a newly arrived task is quickly provided a quantum of service.
In receiver-initiated algorithms, the polling starts when a node becomes a receiver.
However, it is unlikely that these polls will be received at senders when new tasks
that have arrived at them have not yet begun executing. As a result, a drawback of
receiver-initiated algorithms is that most transfers are preemptive and therefore expen­
sive. Conversely, sender-initiated algorithms are able to make greater use of nonpre­
emptive transfers because they can initiate load distributing activity as soon as a new
task arrives.

11.6.3 Symmetrically Initiated Algorithms

Under symmetrically initiated algorithms [21], both senders and receivers search for
receivers and senders, respectively, for task transfers. These algorithms have the ad­
vantages of both sender- and receiver-initiated algorithms. At low system loads, the
sender-initiated component is more successful in finding underloaded nodes. At high
system loads, the receiver-initiated component is more successful in finding overloaded
nodes. However, these algorithms are not immune from the disadvantages of both
sender- and receiver-initiated algorithms. As in sender-initiated algorithms, polling at
high system loads may result in system instability, and as in receiver-initiated algo­
rithms, a preemptive task transfer facility is necessary.

A simple symmetrically initiated algorithm can be constructed by using both the
transfer and location policies described in Secs. 11.6.1 and 11.6.2. Another symmetri­
cally initiated algorithm, called the above-average algorithm [20], is described next.

THE ABOVE-AVERAGE ALGORITHM. The above-average algorithm, proposed by
Krueger and Finkel [20], tries to maintain the load at each node within an acceptable
range of the system average can cause
becoming either a
description of this.

Transfer policy. These thresholds
across all nodes. 
lower threshold =
upper threshold =
within the acceptable

Location policy. The
Sender-initiated compo­
- A sender (a node in TooHigh message)
- A receiver (a node in TooHigh message)
- TooHigh message to be received),
- receiver from over­
- timeout expires 
- is decreased.
- On receiving an
- When a sender
- TooLow message.
- TooLow timeout expires
- does under sende
- TooLow timeout expires
- broadcasts a Cham
- other nodes.
at random and (i.e., a sender) is
a predetermined
node is still a
period, the extra
until another
use the polling ac-
will find a suitable
system loads. At low
s (such as round-
quantum of service.
becomes a receiver.
when new tasks
drawback of
forever use of nonpre-
acceptable
range of the system average. Striving to maintain the load at a node at the exact system average can cause processor thrashing \[3\], as the transfer of a task may result in a node becoming either a sender (load above average) or a receiver (load below average). A description of this algorithm follows.

Transfer policy. The transfer policy is a threshold policy that uses two adaptive thresholds. These thresholds are equidistant from the node's estimate of the average load across all nodes. For example, if a node's estimate of the average load is 2, then the lower threshold = 1 and the upper threshold = 3. A node whose load is less than the lower threshold is considered a receiver, while a node whose load is greater than the upper threshold is considered a sender. Nodes that have loads between these thresholds lie within the acceptable range, so they are neither senders nor receivers.

Location policy. The location policy has the following two components:

Sender-initiated component

- A sender (a node that has a load greater than the acceptable range) broadcasts a TooHigh message, sets a TooHigh timeout alarm, and listens for an Accept message until the timeout expires.
- A receiver (a node that has a load less than the acceptable range) that receives a TooHigh message cancels its TooLow timeout, sends an Accept message to the source of the TooHigh message, increases its load value (taking into account the task to be received), and sets an AwaitingTask timeout. Increasing its load value prevents a receiver from over-committing itself to accepting remote tasks. If the AwaitingTask timeout expires without the arrival of a transferred task, the load value at the receiver is decreased.
- On receiving an Accept message, if the node is still a sender, it chooses the best task to transfer and transfers it to the node that responded.
- When a sender that is waiting for a response for its TooHigh message receives a TooLow message, it sends a TooHigh message to the node that sent the TooLow message. This TooHigh message is handled by the receiver as described under the "Receiver-initiated component."
- On expiration of the TooHigh timeout, if no Accept message has been received, the sender infers that its estimate of the average system load is too low (since no node has a load much lower). To correct this problem, the sender broadcasts a ChangeAverage message to increase the average load estimate at the other nodes.

Receiver-initiated component

- A node, on becoming a receiver, broadcasts a TooLow message, sets a TooLow timeout alarm, and starts listening for a TooHigh message.
- If a TooHigh message is received, the receiver performs the same actions that it does under sender-initiated negotiation (see above).
- If the TooLow timeout expires before receiving any TooHigh messages, the receiver broadcasts a ChangeAverage message to decrease the average load estimate at the other nodes.
Selection policy. This algorithm can make use of any of the approaches discussed under the selection policy in Sec. 11.4.2.

Information policy. The information policy is demand-driven. A highlight of this algorithm is that the average system load is determined individually at each node, imposing little overhead and without the exchange of many messages. Another key point to note is that the acceptable range determines the responsiveness of the algorithm. When the communication network is heavily/lightly loaded (indicated by long/short message transmission delays, respectively), the acceptable range can be increased/decreased by each node individually so that the load balancing actions adapt to the state of the communication network as well.

11.6.4 Adaptive Algorithms

A STABLE SYMMETRICALLY INITIATED ALGORITHM. The main cause of system instability due to load sharing by the previous algorithms is the indiscriminate polling by the sender's negotiation component. The stable symmetrically initiated algorithm [31] utilizes the information gathered during polling (instead of discarding it as was done by the previous algorithms) to classify the nodes in the system as either Sender/overloaded, Receiver/underloaded, or OK (i.e., nodes having manageable load). The knowledge concerning the state of nodes is maintained by a data structure at each node, comprised of a senders list, a receivers list, and an OK list. These lists are maintained using an efficient scheme in which list manipulative actions, such as moving a node from one list to another, or finding the list to which a node belongs, impose a small and constant overhead irrespective of the number of nodes in the system. (See [31] for more details on the list maintenance scheme.)

Initially, each node assumes that every other node is a receiver. This state is represented at each node by a receivers list that contains all nodes (except itself), an empty senders list, and an empty OK list.

Transfer policy. The transfer policy is a threshold policy where decisions are based on CPU queue length. The transfer policy is triggered when a new task originates or when a task departs. The transfer policy makes use of two threshold values to classify the nodes: a lower threshold (LT) and an upper threshold (UT). A node is said to be a sender if its queue length > UT, a receiver if its queue length < LT, and OK if LT ≤ node’s queue length ≤ UT.

Location policy. The location policy has the following two components:

Sender-initiated Component. The sender-initiated component is triggered at a node when it becomes a sender. The sender polls the node at the head of the receivers list to determine whether it is still a receiver. The polled node removes the sender node ID from the list it is presently in, puts it at the head of its senders list, and informs the sender whether it is a receiver, sender, or OK node based on its current status. On receipt of this reply, the sender transfers the new task if the polled node has indicated that it is a receiver. Otherwise, the polled node’s ID is removed from the receivers list and put at the head of the OK list or at the head of senders list based on its reply. Then the sender polls the node at the head of the receivers list.

Receiver-initiated Component. The receiver-initiated component is triggered at a node when it becomes a receiver. The receiver polls the node at the head of the senders list to determine whether it is still a sender. The polled node removes the receiver node’s ID from the list it is presently in, puts it at the head of its receivers list, and informs the receiver whether it is a receiver, sender, or OK node based on its current status. On receipt of this reply, the receiver transfers the new task if the polled node has indicated that it is a sender. Otherwise, the polled node’s ID is removed from the senders list and put at the head of the OK list or at the head of receivers list based on its reply. Then the receiver polls the node at the head of the senders list.
The polling process stops if a suitable receiver is found for the newly arrived task, if the number of polls reaches a PollLimit (a parameter of the algorithm), or if the receivers list at the sender node becomes empty. If polling fails to find a receiver, the task is processed locally, though it can later migrate as a result of receiver-initiated load sharing.

**Receiver-initiated Component.** The goal of the receiver-initiated component is to obtain tasks from a sender node. The nodes polled are selected in the following order: head to tail in the senders list (the most up-to-date information is used first); then tail to head in the OK list (the most out-of-date information is used first, in the hope that the node has become a sender); then tail to head in the receivers list (again the most out-of-date information is used first).

The receiver-initiated component is triggered at a node when the node becomes a receiver. The receiver polls the selected node to determine whether it is a sender. On receipt of the message, the polled node, if it is a sender, transfers a task to the polling node and informs it of its state after the task transfer. If the polled node is not a sender, it removes the receiver node ID from the list it is presently in, puts it at the head of its receivers list, and informs the receiver whether it (the polled node) is a receiver or OK. On receipt of the reply, the receiver node removes the polled node ID from whatever list it is presently in and puts it at the head of the appropriate list based on its reply.

The polling process stops if a sender is found, if the receiver is no longer a receiver, or if the number of polls reaches a static PollLimit.

**Selection policy.** The sender-initiated component considers only newly arrived tasks for transfer. The receiver-initiated component can make use of any of the approaches discussed under the selection policy in Sec. 11.4.2.

**Information policy.** The information policy is demand-driven, as the polling activity starts when a node becomes a sender or a receiver.

**Discussion.** At high system loads, the probability of a node being underloaded is negligible, resulting in unsuccessful polls by the sender-initiated component. Unsuccessful polls result in the removal of polled node IDs from receivers lists. Unless receiver-initiated polls to these nodes fail to find them as senders, which is unlikely at high system loads, the receivers lists remain empty. As a result, future sender-initiated polls at high system loads (which are most likely to fail) are prevented. (Note that a sender polls only nodes found in its receivers list.) Hence, the sender-initiated component is deactivated at high system loads, leaving only receiver-initiated load sharing (which is effective at such loads).

At low system loads, receiver-initiated polling generally fails. These failures do not adversely affect performance because extra processing capacity is available at low system loads. In addition, these polls have the positive effect of updating the receivers lists. With the receivers lists accurately reflecting the system state, future sender-initiated load sharing will generally succeed within a few polls. Thus, by using sender-initiated load sharing at low system loads, receiver-initiated load sharing at high loads, and symmetrically initiated load sharing at moderate loads, the stable symmetrically initiated
algorithm achieves improved performance over a wide range of system loads while preserving system stability.

A STABLE SENDER-INITIATED ALGORITHM. This algorithm [31] has two desirable properties. First, it does not cause instability. Second, load sharing is due to non-preemptive transfers (which are cheaper) only. This algorithm uses the sender-initiated load sharing component of the stable symmetrically initiated algorithm as is, but has a modified receiver-initiated component to attract the future nonpreemptive task transfers from sender nodes. The stable sender-initiated policy is very similar to the stable symmetrically initiated approach, so only the differences will be pointed out.

In the stable sender-initiated algorithm, the data structure (at each node) of the stable symmetrically initiated algorithm is augmented by an array called the statevector. The statevector is used by each node to keep track of which list (senders, receivers, or OK) it belongs to at all the other nodes in the system. Moreover, the sender-initiated load sharing is augmented with the following step: when a sender polls a selected node, the sender’s statevector is updated to reflect that the sender now belongs to the senders list at the selected node. Likewise, the polled node updates its statevector based on the reply it sent to the sender node to reflect which list it will belong to at the sender.

The receiver-initiated component is replaced by the following protocol: when a node becomes a receiver, it informs all the nodes that are misinformed about its current state. The misinformed nodes are those nodes whose receivers lists do not contain the receiver’s ID. This information is available in the statevector at the receiver. The statevector at the receiver is then updated to reflect that it now belongs to the receivers list at all those nodes that were informed of its current state. By this technique, this algorithm avoids receivers sending broadcast messages to inform other nodes that they are receivers. Remember that broadcast messages impose message handling overhead at all nodes in the system. This overhead can be high if nodes frequently change their state.

Note that there are no preemptive transfers of partly executed tasks here. The sender-initiated load sharing component will perform any load sharing, if possible on the arrival of a new task. The stability of this approach is due to the same reasons given for the stability of the stable symmetrically initiated algorithm.

11.7 PERFORMANCE COMPARISON

This section discusses the general performance trends of some of the example algorithms described in the previous section. Figure 11.5 through Fig. 11.7 plot the average response time of tasks vs. the offered system load for several load sharing algorithms discussed in Sec. 11.6 [32]. The average service demand for tasks is assumed to be one time unit, and the task interarrival times and service demands are independently exponentially distributed. The system load is assumed to be homogeneous; that is, all nodes have the same long-term task arrival rate. The system is assumed to contain 40 identical nodes. The notations used in the figures correspond to the algorithms as follows:

| M/M/K | A fixed threshold technique.
| RECV | Receivers use load distributing random location policy.
| RAND | Senders use load distributing random location policy.
| SEND | Senders use load distributing random location policy.
| ADSEND | Senders use adaptive load distributing random location policy.
| SYM | Senders use load distributing symmetric location policy.
| ADSYM | Senders use adaptive load distributing symmetric location policy.

A fixed threshold technique is used for task transfer cost. The system is homogeneous, with a load of 1 on each node. The statevector at the receiver is updated to reflect that it now belongs to the receivers list at all those nodes that were informed of its current state. By this technique, this algorithm avoids receivers sending broadcast messages to inform other nodes that they are receivers. Remember that broadcast messages impose message handling overhead at all nodes in the system. This overhead can be high if nodes frequently change their state.

For these comparisons, such a small limit is below threshold, so the probability that a node becomes a receiver decreases on the first pass. However, since most nodes exceed on the first pass, the total response time that will be spent in this search after the first pass is very small.

Main result. Comparing the receiver-initialized scheme with the symmetric location policy, the receiver-initialized scheme provides a significant improvement in response time because it uses load distributing through simple sender-initiated schemes. M/M/K gives a slight improvement over the other schemes.

11.7.1 Receiver-initialized schemes

It can be observed from the figures that the receiver-initialized schemes are marginally better than...
A distributed system that performs no load distributing.
RECV
Receiver-initiated algorithm.
RAND
Sender-initiated algorithm with random location policy.
SEND
Sender-initiated algorithm with threshold location policy.
ADSEND
Stable sender-initiated algorithm.
SYM
Symmetrically initiated algorithm (SEND and RECV combined).
ADSYM
Stable symmetrically initiated algorithm.
M/M/K
A distributed system that performs ideal load distributing without incurring any overhead.

A fixed threshold of \( T = \text{lower threshold} = \text{upper threshold} = 1 \) was used for these comparisons. However, the value of \( T \) should adapt to the system load and the task transfer cost because a node is identified as a sender or a receiver by comparing its queue length with \( T \). At low system loads, many nodes are likely to be idle—a low value of \( T \) will result in nodes with small queue lengths being identified as senders who can benefit by transferring load. At high system loads, most nodes are likely to be busy—a high value of \( T \) will result in the identification of only those nodes with significant queue lengths as senders, who can benefit the most by transferring load. While a scheduling algorithm may adapt to the system load by making use of an adaptive \( T \), the adaptive stable algorithms of Sec.11.6.4 adapt to the system load by varying the PollLimit with the help of the lists. Also, low thresholds are desirable for low transfer costs as smaller differences in node queue lengths can be exploited; high transfer costs demand higher thresholds.

For these comparisons, a small, fixed PollLimit = 5 was assumed. We can see why such a small limit is sufficient by noting that if \( P \) is the probability that a particular node is below threshold, then (because the nodes are assumed to be independent) the probability that a node below threshold is first encountered on the \( i \)th poll is \( P(1 - P)^{i-1} \). For large \( P \), this expression decreases rapidly with increasing \( i \); the probability of succeeding on the first few polls is high. For small \( P \), the quantity decreases more slowly. However, since most nodes are above threshold, the improvement in systemwide response time that will result from locating a node below threshold is small; quitting the search after the first few polls does not carry a substantial penalty.

Main result. Comparing M/M/1 with the sender-initiated algorithm that uses the random location policy (RAND) in Fig. 11.5, we see that even this simple load distributing scheme provides a substantial performance improvement over a system that does not use load distributing. Considerable further improvement in performance can be gained through simple sender-initiated (SEND) and receiver-initiated (RECV) load sharing schemes. M/M/K gives the optimistic lower bound on the performance that can be obtained through load distributing, since it assumes no load distributing overhead.

11.7.1 Receiver-initiated vs. Sender-initiated Load Sharing

It can be observed from Fig. 11.5 that the sender-initiated algorithm (SEND) performs marginally better than the receiver-initiated algorithm (RECV) at light to moderate
system loads, while the receiver-initiated algorithm performs substantially better at high system loads. Receiver-initiated load sharing is less effective at low system loads because load sharing is not initiated when one of the few nodes becomes a sender, and thus load sharing often occurs late.

Regarding the robustness of these policies, the receiver-initiated policy has an edge over the sender-initiated policies. The receiver-initiated policy performs acceptably with a single value of the threshold over the entire system load spectrum, whereas the sender-initiated policy requires an adaptive location policy to perform acceptably at high loads. It can be seen from Fig. 11.5 that at high system loads, the receiver-initiated policy maintains system stability because its polls generally find busy nodes, while polls due to the sender-initiated policy are generally ineffective and waste resources in efforts to find underloaded nodes.

### 11.7.2 Symmetrically Initiated Load Sharing

This policy takes advantage of its sender-initiated load sharing component at low system loads, its receiver-initiated component at high system loads, and both of these components at moderate system loads. Hence, its performance is better or matches that of the sender-initiated policy at all levels of system load, and is better than that of receiver-initiated policy at low to moderate system loads [32] (Fig. 11.6). Nevertheless, this policy also causes system instability at high system loads because of the ineffective polling activity of its sender-initiated component at such loads.

### 11.7.3 Stable Load Sharing

The performance of the receiver-initiated policy is better than that of M/M/K (Fig. 11.5) since it assumes no load preemption. The sender-initiated policy also performs better than M/M/K at high loads (> 0.7), but the improvement is the same as that of the receiver-initiated policy. Furthermore, this policy does not start in accordance with a subset of nodes and random polling. As a result, nodes are polled randomly, causing system instability.

### 11.7.4 Performance of Heterogeneous Workloads

Figure 11.8 plots the average response time vs. system load at a constant offered workload, while the other figure, we observe that the performance of the receiver-initiated policy is better than any other algorithm with low load preemption. The system does not start in accordance with a subset of nodes, and random polling. As fewer nodes are involved in load sharing, the performance improves.
11.7.3 Stable Load Sharing Algorithms

The performance of the stable symmetrically initiated algorithm (ADSYM) approaches that of M/M/K (Fig. 11.7), though this optimistic lower bound can never be reached, as it assumes no load distributing overhead. The performance of ADSYM matches that of the sender-initiated algorithm at low system loads and offers substantial improvements at high loads (> 0.85) over all the nonadaptive algorithms [31]. This performance improvement is the result of its judicious use of the knowledge gained by polling. Furthermore, this algorithm does not cause system instability.

The stable sender-initiated algorithm (ADSEND) yields a better performance than the unstable sender-initiated policy (SEND) for system loads > 0.6 and does not cause system instability. While ADSEND is not as effective as ADSYM, it does not require expensive preemptive task transfers.

11.7.4 Performance Under Heterogeneous Workloads

Heterogeneous workloads have been shown to be common for distributed systems [19]. Figure 11.8 plots mean response time against the number of nonload generating nodes at a constant offered system load of 0.85. These nodes originate none of the system workload, while the remaining nodes originate all of the system workload. From the figure, we observe that RECV becomes unstable at a much lower degree of heterogeneity than any other algorithm. The instability occurs because, in RECV, the load sharing does not start in accordance with the arrivals of tasks at a few (but highly overloaded) sender nodes, and random polling by RECV is likely to fail to find a sender when only a small subset of nodes are senders. SEND also becomes unstable with increasing heterogeneity. As fewer nodes receive all the system load, it is imperative that they quickly transfer
tasks. But the senders become overwhelmed, as random polling is ineffective in reducing wasteful tries. SYM also becomes unstable at higher levels of heterogeneity because of ineffective polling. SYM outperforms RECV and SEND because it can transfer tasks at a higher rate than either RECV or SEND alone can. The sender-initiated algorithm with the random location policy (RAND) performs better than most algorithms at extreme levels of heterogeneity. By simply transferring tasks from the load-generating nodes to randomly selected nodes without any regard to their status, it essentially balances the load across all nodes in the system, thus avoiding instability.

Only ADSYM remains stable and performs better with increasing heterogeneity. As heterogeneity increases, senders rarely change their state and will generally be in senders list at nonload generating nodes. The nonload generating nodes will alternate between OK and receiver states and appear in OK or receivers lists at senders. When the lists accurately represent the system state, nodes are often successful at finding partners.

### 11.8 SELECTING A SUITABLE LOAD SHARING ALGORITHM

Based on the performance trends of load sharing algorithms, one may select a load sharing algorithm that is appropriate to the system under consideration as follows:

1. If the system under consideration never attains high loads, sender-initiated algorithms will give an improved average response time over no load sharing at all.

2. Stable scheduling algorithms are recommended for systems that can reach high loads. These algorithms perform better than nonadaptive algorithms for the following reasons:

- Under sender-initiated messages deliv-
ors are advel
the perform-
most inquiries
- Receiver-initi-
of preemptive
compared to r
municating a

3. For a system that
ratorlarly initiated s
performance and

4. For a system that
migration of
ommended, as the
loads, perform better
are stable at high

5. For a system the
rrithms are prefered
nonadaptive algn
The system load vs. effective in reducing heterogeneity because of can transfer tasks at a suitable algorithm with algorithms at extreme generating nodes to generally balances the using heterogeneity. will generally be in nodes will alternate at finding partners. When the selecting a load solution as follows:

• Under sender-initiated algorithms, an overloaded processor must send inquiry messages delaying the existing tasks. If an inquiry fails, two overloaded processors are adversely affected because of unnecessary message handling. Therefore, the performance impact of an inquiry is quite severe at high system loads, where most inquiries fail.

• Receiver-initiated algorithms remain effective at high loads but require the use of preemptive task transfers. Note that preemptive task transfers are expensive compared to nonpreemptive task transfers because they involve saving and communicating a far more complicated task state.

3. For a system that experiences a wide range of load fluctuations, the stable symmetrically initiated scheduling algorithm is recommended because it provides improved performance and stability over the entire spectrum of system loads.

4. For a system that experiences wide fluctuations in load and has a high cost for the migration of partly executed tasks, stable sender-initiated algorithms are recommended, as they perform better than unstable sender-initiated algorithms at all loads, perform better than receiver-initiated algorithms over most system loads, and are stable at high loads.

5. For a system that experiences heterogeneous work arrival, adaptive stable algorithms are preferable, as they provide substantial performance improvement over nonadaptive algorithms.

FIGURE 11.8
Average response time vs. number of load generating machines (adapted from [32]).