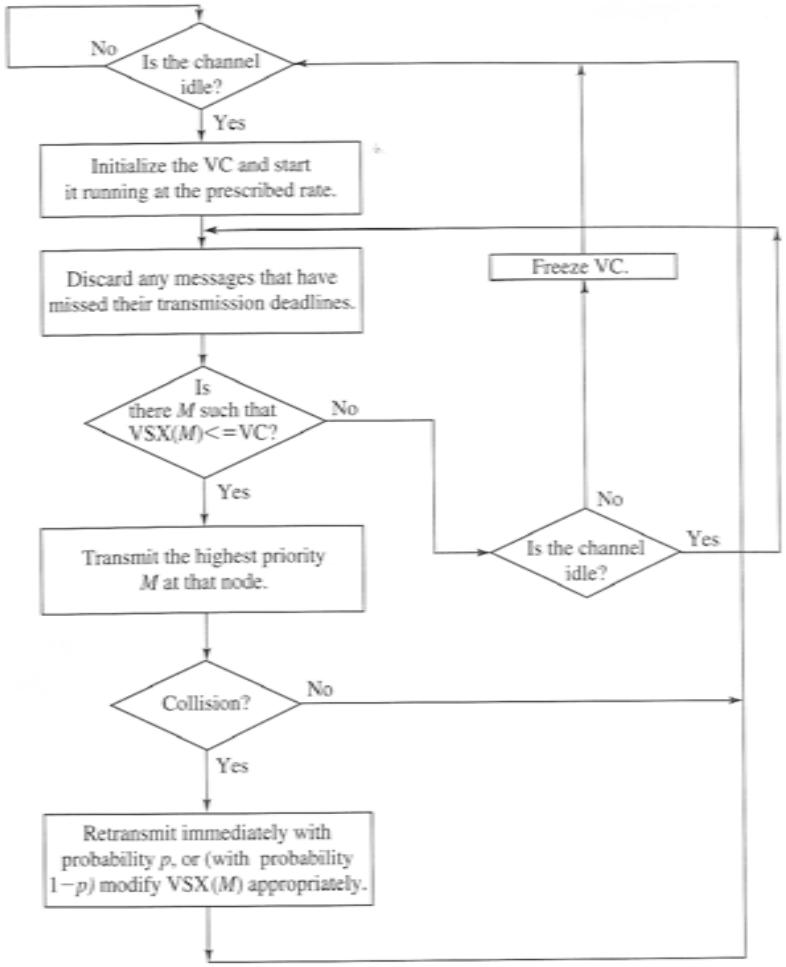

FIGURE 6.19
Flowchart of the VTCSMA algorithm.

It is easy to show that VTCSMA-A, VTCSMA-T, VTCSMA-D, and VTCSMA-L implement the earliest-arrival-first, the minimum-transmission-time-first, the earliest-deadline, and the minimum-laxity priority algorithms, respectively. The VC is consistent (plus or minus a small skew) for all the nodes, and the transmission is based on the relationship between VC and VSX(M) for each packet M.

Example 6.9. As an example of how these algorithms work, consider the VTCSMA-L algorithm. Let $\eta = 2$ (i.e., the VC runs twice as fast as the RC when the channel is idle). Let us assume that the transmission time for each packet is $T_M = 15$, and that the propagation time is $\tau = 1$. Suppose the packets arrive according to the following table:

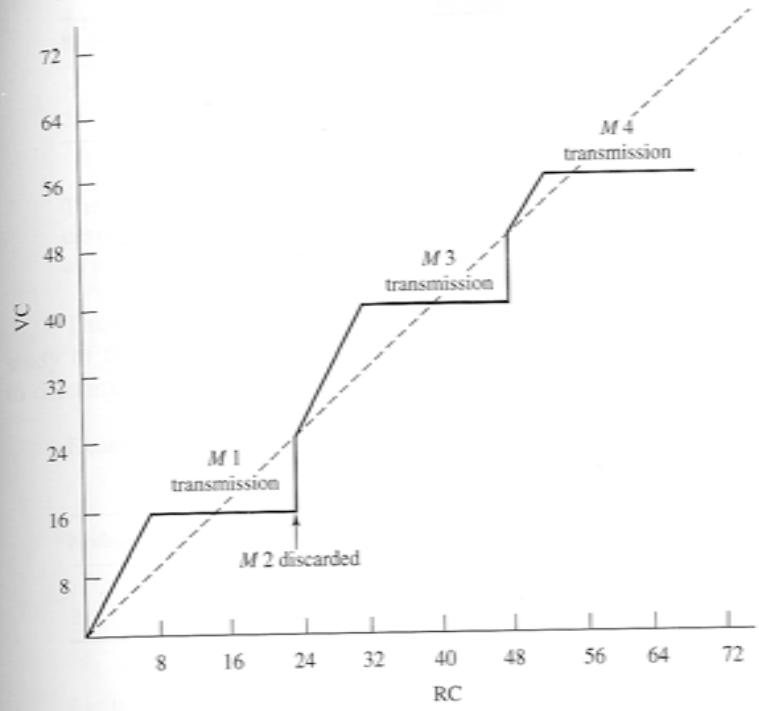

FIGURE 6.20
VC-RC trajectory for Example 6.9: $\eta = 2$.

Node	M	RC at arrival	D_M	L_M
1	1	0	32	16
2	2	10	36	20
3	3	20	56	40
4	4	20	72	56

Figure 6.20 summarizes what happens. M1 starts transmission at RC = 8 (when VC = L_1) and completes at RC = 8 + 16 = 24. This is too late for M2 to start transmitting, and so it is discarded. The VC is initialized at 24 and is restarted. When it reaches $L_3 = 40$, 1M3 starts transmitting. And so on.

Note that despite there being sufficient time to transmit all four packets successfully, M2 had to be discarded. This is because the channel was needlessly idle from RC = 0 to RC = 8. This happened because η was not sufficiently large.

Let us see what happens if we make $\eta = 4$. Figure 6.21 tells the story—all four packets get successfully transmitted.

Does this mean that the larger the value of η, the better the performance of the system? Not necessarily. Consider the following example. Once again, the algorithm is VTCSMA-L.

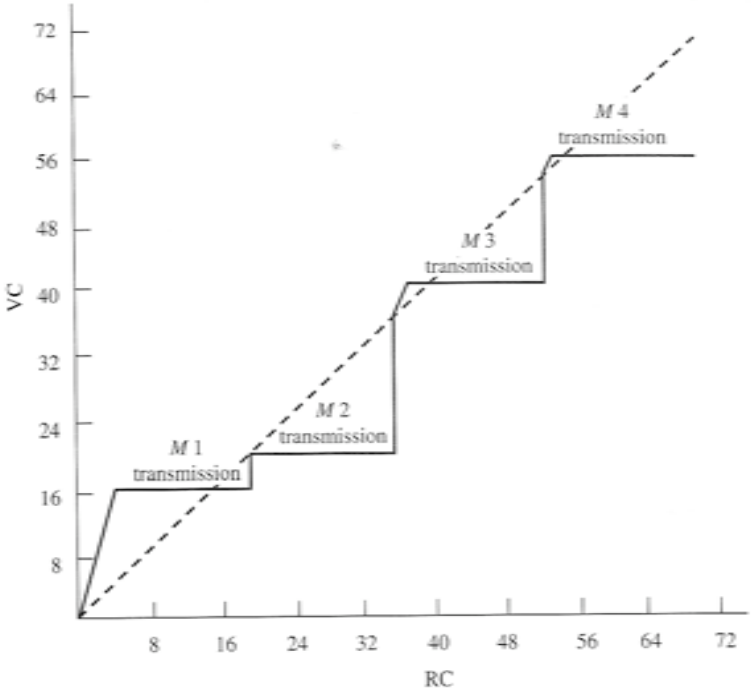

FIGURE 6.21
VC-RC trajectory for Example 6.9: $\eta = 4$.

Example 6.10. Let the parameters be as in the following table.

Node	M	RC at arrival	D_M	L_M
1	1	6	22	6
2	2	6	40	24

Let the other parameters be as in Example 6.9. We encourage the reader to draw the VC-RC trajectory for the cases $\eta = 2$ and $\eta = 4$. For $\eta = 2$, both packets are successfully transmitted; for $\eta = 4$, M1 and M2 collide and, as a result, M1 has to be discarded since there is not time enough after the collision is resolved to successfully transmit it.

The reader will have noticed that in Example 6.9, the channel is needlessly idle for some time even when it has a packet awaiting transmission. This is to accomodate later arrivals with tighter deadline requirements, as in Example 6.11.

Example 6.11. The following table lists some packet arrivals. Let $\eta = 2$ and the other parameters be as in Example 6.9.

Node	M	RC at arrival	D_M	L_M
1	1	0	40	24
2	2	5	24	8

The reader can easily check that keeping the channel idle over the interval [0, 5], despite M1 awaiting transmission, has enabled M2, which arrived at RC = 5 but had an earlier deadline, to be transmitted successfully.

Thus far, we have assumed clock skew to be negligible. No systematic study of the effects of clock skew have been carried out, but its effects are easy to demonstrate.

Example 6.12. Consider again VTCSMA-L, and the following arrival pattern.

Node	M	Actual RC at arrival	D_M	L_M
1	1	8	32	16
2	2	9	36	20

Let the clocks be skewed with respect to the actual real time as follows.

Node	RC at node
1	Actual RC − 1
2	Actual RC + 1

We leave it to the reader to show that for $\eta = 2$, this will cause nodes 1 and 2 to transmit their respective packets on the network at the same time, causing a collision and also ensuring that M1 cannot be successfully transmitted.

Clock skew is not always deleterious. To see this, the reader should consider the packet arrival times in Example 6.9, and develop a set of skews for which all four packets are successfully transmitted for $\eta = 2$.

Performance of the VTCSMA algorithm. No analytical model exists to calculate the fraction of packets that miss their deadline under the VTCSMA algorithm. However, simulation studies have been carried out by Zhao and Ramamritham [22], and the following conclusions can be drawn from them.

1. VTCSMA protocols have a better loss rate (i.e., rate at which deadlines are missed), in general, than the CSMA protocols.
2. The best performance in terms of fraction of loss rate is obtained by the VTCSMA-D algorithm.

3. The performance is a function of the value of η; however, as long as the transmission delays are not too great and the network is not overloaded, the range of η values over which each protocol performs close to its best is large.

WINDOW PROTOCOL. Like the VTCSMA protocol, the window protocol is based on collision sensing. Again, it cannot be guaranteed that messages will be transmitted in time to meet their deadlines. This protocol is therefore only suitable for soft real-time systems.

As with VTCSMA, the system consists of a set of nodes connected on a bus. Each node continuously monitors the bus to receive any messages that may be addressed to it. Since all activity on the bus is equally visible to all the nodes, we can assume that each node knows when a transmission has succeeded and when multiple simultaneous transmissions collide, even if the transmissions in question neither originate from, nor are destined for, that node. Events on the bus thus become a mechanism for synchronizing the actions of the nodes.

The protocol owes its name to the window maintained at each node. The window is a time interval, and the windows of all the nodes are identical. When the latest-time-to-transmit (LTTT) of a packet falls within this window and the channel is idle, the packet is eligible to be transmitted. If more than one packet is eligible for transmission at a node, one of them is picked based on some criterion (e.g., LTTT).

We assume that the clock skews are fairly small (i.e., the nodes are tightly synchronized). The time axis is broken down into *slots*, where each slot is an interval of time equal to the end-to-end network propagation time. A node may begin transmitting a packet only at the beginning of a time slot.

Figures 6.22 to 6.25 describe the window algorithm. We use the following notation:

δ	Predefined integer constant.
t	Present time.
$LTTT_M$	LTTT associated with message M.
`Random(a,b)`	Random number, distributed uniformly between `a` and `b`.
T_M	Number of slots needed to transmit message M.

The algorithm maintains a stack in each node to record the window history. Each stack entry is a two-tuple `(u,i)`. The `u` field reflects the upper bound of a window in which a collision occurred, and the `i` field is zero unless the node has a message involved in the collision; in such an event, `i` contains the ID of that message. We can informally describe the protocol as follows. Central to the algorithm is the window, which is a duration of time (in integral multiples of slots). Each node in the system has exactly the same window. The initial window size is δ. The window is modified based on events occurring on the bus. Since the bus is continually monitored by all the nodes, all the nodes can maintain, by means of a distributed algorithm, an identical copy of the window.

If a node has a packet to transmit, it checks if there is an ongoing transmission on the bus. If there is, it waits until that transmission ceases; this can be


```
Initialization:
   up := t + δ;
   empty the stack;
   If one or more messages have LTTT in the interval [t,up), transmit
         the one with minimum LTTT.

At the beginning of each slot:
   Find out if there is any message with (LTTT < t),
         and drop it, since it has missed its deadline.
   Discard any stack contents whose u field is less than t;
   If the channel is busy due to a collision, then
         abort any ongoing transmission by this node;
   else if the channel is idle following a collision
         contract_window_and_send(up, t);
   else if the channel is occupied by a message transmission
         continue the transmission, if any;
   else if the channel is idle after a successful transmission, then
         pop_and_send(up, t);
   else if the channel continues idle, then
         expand_window_and_send(up, t);
   end if;
end;
```

FIGURE 6.22
Window protocol. (W. Zhao, et al., "A Window Protocol for Transmission of Time-Constrained Messages," *IEEE Trans. Computers*, Vol. 39, No. 9, 1990. ©IEEE 1990. Reprinted with permission.)

detected by noting that the bus has been idle for at least one slot. The node then transmits its message on the channel at the beginning of the next slot provided that the LTTT of that message is within the current window. If more than one node starts transmitting in the same slot, the packets collide. This collision is detected, and both transmissions are aborted. The nodes now cause the window to contract, and then only those messages that have LTTTs within the contracted window are transmitted. If there is only one such message, the transmission is successful. If there is more than one message, there is another collision and the window contracts again.

If the contraction of the window is followed by silence on the bus, it means that none of the packets has an LTTT within the current window. The window must then either be expanded or translated to the right.

If there is a collision despite the window's shrinking to one slot, it means that there are multiple packets with the same LTTT. Then, each node

- with probability p, retransmits its message in the next slot (p is a design parameter set by the user or designer), or

```
Procedure contract_window_and_send(up, t);
BEGIN
    if (up > t) then
        if (up > t+1) then
            if this node was involved in the collision then
                i = identifier of its message that collided;
            else
                i = 0;
            end if;
            push (up,i) onto the stack;
            up = t + ⌈ (up - t)/2 ⌉;
            send out the message if its LTTT is in [t,up);
        else
            if (this node was involved in the collision) then
                if (Random(0,1) > P)
                    retransmit message that collided;
                else
                    if (t is the latest time to transmit
                         the message to meet its deadline) then
                        discard the message that collided;
                    else
                        LTTT_M = Random(t+2, D_M-T_M);
                            /*M is the collided message*
                    end if;
                end if;
            end if;
        end if;
    else
        pop_and_send(up,t);
    end if;
END;
```

FIGURE 6.23
Procedure contract_window_and_send. (W. Zhao, et al., "A Window Protocol for Transmission of Time-Constrained Messages," *IEEE Trans. Computers*, Vol. 39, No. 9, 1990. © IEEE 1990. Reprinted with permission.)

- with probability $(1 - p)$, it does not retransmit in the next slot, but reassigns to the message a new randomly-generated LTTT (which will still enable the message deadline to be met). If this is not possible (i.e., the message must be transmitted in this slot or it will miss its deadline), the message is discarded.

At the end of this operation, the window is also modified suitably.

Correctly managing the window size is central to the success of this algorithm. If the size is very small, the likelihood increases that the channel will be


```
Procedure pop_and_send(up, t);
BEGIN
    if the stack is non-empty, then
        Pop the stack and set (up = u field of popped item)
    else
        up = max(up, t) + δ;
    end if;
    transmit the message with smallest LTTT
        if its LTTT is in [t,up);
END
```

FIGURE 6.24
Procedure pop_and_send. (W. Zhao, et al., "A Window Protocol for Transmission of Time-Constrained Messages," *IEEE Trans. Computers*, Vol. 39, No. 9, 1990. ©IEEE 1990. Reprinted with permission.)

```
Procedure expand_window_and_send(up, t);

BEGIN
    if the stack is empty, then
        up = t + δ;
        Send out message, if any, with minimum LTTT which is in [t,up);
    else
        u_top = the u field of the top item in the stack;
        i_top = the i field of the top item in the stack;
        if (up < u_top - 1 ) then
            up = ⌈ (up + u_top)/2 ⌉;
            Send out message, if any, with minimum LTTT which is in [t,up);
        else /* there is a tie between the LTTT of two messages*/
            if there is a message M in the queue with
            identifier equal to i_top, then
            if Random(0,1) > P
                transmit M;
            else
                set LTTT_M = Random(t + 2, D_M - τ_M);
            end if;
        end if;
    Pop the stack and set up = u field of popped item.
END; \ end if;
```

FIGURE 6.25
Procedure expand_window_and_send. (W. Zhao, et al., "A Window Protocol for Transmission of Time-Constrained Messages," *IEEE Trans. Computers*, Vol. 39, No. 9, 1990. ©IEEE 1990. Reprinted with permission.)

idle even though nodes have packets to transmit. If it is too large, the likelihood of collisions increases.

Example 6.13. Let us consider the performance of this algorithm for a system consisting of three nodes, N_1, N_2, and N_3. We use $\delta = 20$, and assume each message takes one slot to transmit. Assume that, when the algorithm starts, nodes N_1, N_2, N_3 each has one packet awaiting transmission with LTTT values 19, 19, and 3, respectively. The identifier of each of these packets is 1. No other packets arrive over the first dozen slots.

At the beginning of slot 0, the window is set by each node to [0,20). Since each node has a packet in this window, they all attempt to transmit. There is a collision and it is detected by the end of the slot. In slot 1, the channel is idle after a collision. At the beginning of slot 2, each node contracts its window to [2,11). The three nodes each push (20,1) into their respective stacks. Only N_3 has a packet whose LTTT falls within this new window, and it transmits successfully. This transmission is completed by the end of slot 2, and the channel is idle following a successful transmission during slot 3.

At the beginning of slot 4, the nodes run the pop_and_send algorithm. The value of up is changed from 11 back to 20. The window is now [4,20). Nodes N_1 and N_2 both attempt to transmit and then collide. The channel is idle following another collision during slot 5.

At the beginning of slot 6, the nodes run the contract_window_and_send algorithm. The previous value of up is pushed into the stack, and the new value is calculated as $6+\lceil(20-6)/2\rceil = 13$. Neither N_1 nor N_2 can transmit in this window, and so the channel continues to be idle during slot 6. At the beginning of slot 7, the nodes run the expand_window_and_send algorithm. Since $\text{up} = 13 < 20-1$, the value of up is reset to $\lceil(13+20)/2\rceil = 17$. The window is now [7,17); N_1 and N_2 still cannot transmit.

At the beginning of slot 8, the nodes run the expand_window_and_send algorithm again. The value of up is now changed to $\lceil(17+20)/2\rceil = 19$. Once again, since the window is [8,19), no transmission takes place (note that the window is open at the right, i.e., the LTTT of 19 is not covered by it).

At the beginning of slot 9, the nodes run the expand_window_and_send algorithm again. However, since $\text{up} = 20-1$, we now detect a tie in LTTT values. Suppose N_1 goes ahead and transmits during this interval while N_2 resets its LTTT to 16. The channel is idle following a successful transmission during slot 10. At the beginning of slot 11, the pop_and_send algorithm is run by each node. This results in N_2 transmitting successfully.

Now, suppose that δ is 10 instead of 20. In such a case, the initial window is [0,10), and N_3 is successful in transmitting during slot 0. The reader is invited to list what happens following this slot in this case.

Performance of the window algorithm. No analytical model of the window algorithm has yet been published. However, some simulation studies were carried out for a Poisson message arrival process, and the following conclusions can be drawn from them:[2]

[2]Keep in mind that these conclusions may or may not be valid for non-Poisson processes.

- The window algorithm was compared to an idealized centralized algorithm, which featured a central controller, with instantaneous knowledge of the state of each node (e.g., the contents of each buffer, etc.). Centralization does away with the collisions, since the central controller decides which node is to transmit at any one time. However, since the overheads associated with the collection of the status information and with the dissemination of the control instructions are assumed to be zero, this centralized algorithm is not realizable in practice. The simulation studies indicate that the window protocol performs close to that of the centralized algorithm.
- It appears that the window protocol is relatively insensitive to the value of δ.
- As with other collision-detection protocols, this protocol exhibits a deadline anomaly: If the deadlines of some messages are relaxed, this can actually worsen system performance.

DISCUSSION. Contention protocols are best when the traffic is light, and the ratio

$$\frac{\text{End-to-end transmission delay}}{\text{Time to transmit a bit}}$$

is low. If the traffic is heavy, then the probability of collisions (and thus the number of aborted transmissions) increases, and bandwidth is wasted. If the delay-to-bit-transmission-time ratio is high, a large number of bits may have been transmitted by the time a collision is detected. A packet affected by a collision has to be retransmitted in its entirety, and so any bits transmitted before a collision is detected waste the network bandwidth.

6.3.2 Token-based Protocols

A *token* is a grant of permission to a node to transmit its packets on the network. When the token-holding node completes its transmission, it surrenders the token to another node. A node is only permitted to transmit on the network if it currently holds the token.

Example 6.14. Token protocols are typically run on buses or rings. An example of a typical ring structure is shown in Figure 6.26. It consists of two rings, with one carrying traffic clockwise and the other counterclockwise. When everything is operational, we have two independent rings operating. If a link on the ring fails, we can reconfigure what is left into a single ring; see Figure 6.27. Similarly, there is the capability to bypass a failed or powered-down node.

Each ring has a token circulating in the appropriate direction. When a node receives the token, it is allowed to start transmitting its messages. It puts them out on the ring. Every other node's network interface receives and then retransmits the packet. When the packet returns to the sender, this node then removes as much of it from the ring as it can. Any remaining fragments of the packet are removed by whichever node currently has the token.

The reason that a node cannot always remove its entire packet from the ring is that it is not until after it has read the source address field that it knows that it was the sender of the packet. By that time, it may have retransmitted the earlier

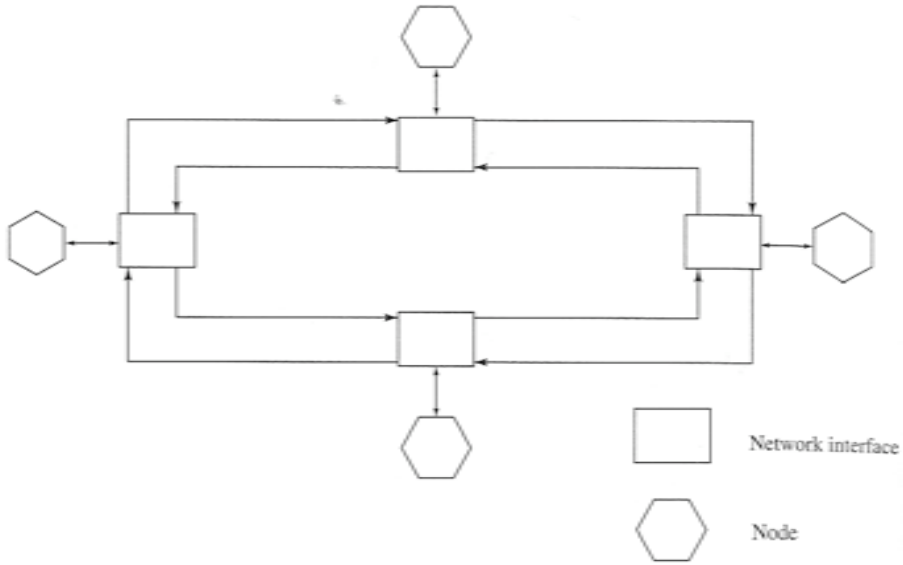

FIGURE 6.26
Multiring network.

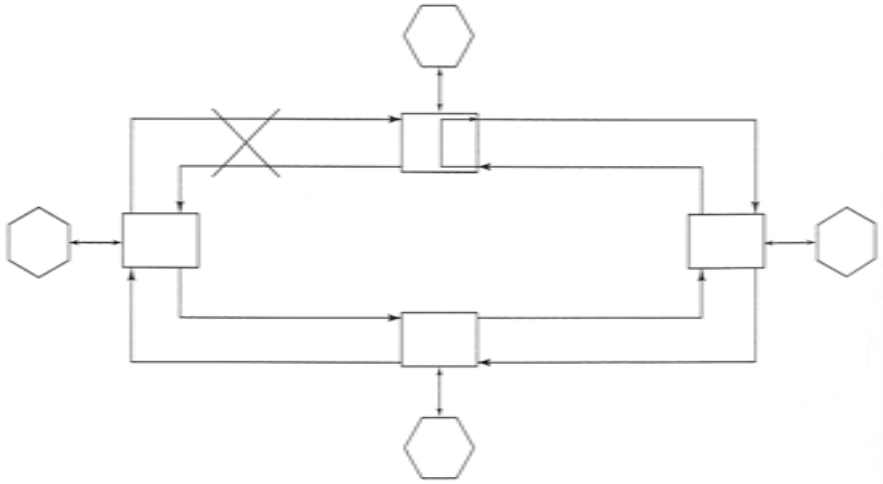

FIGURE 6.27
Dealing with failure in a multiring network.

bits of the packet—this constitutes the remaining fragment that is then removed by the current token-holding node. A token-holding node removes from the ring all the packets that it receives. Thus, if the sender of a packet is still holding the token when one of its packets returns to it, it will remove that packet entirely.

Token algorithms incur the following overheads:

Medium propagation delay: It takes a certain time for a message to propagate from one node to the next.

Token transmission time: Sending out the token takes some time. Since the token is usually much smaller than a frame that contains information, this overhead is typically very small.

Token capture delay: There is usually some time lag between when a node captures the token and when it begins transmitting.

Network interface latency: At each network interface, the input is retransmitted to the output (except for packets that are removed from the ring). The network interface latency is the time between when a bit is received by the network interface and when it is retransmitted.

Token-based protocols are better suited to optical networks than collision-sensing because the ratio of the end-to-end delay time to the time taken in putting out a packet on the ring is large in such networks.

TIMED-TOKEN PROTOCOL. The timed-token protocol is a simple mechanism by which each node is guaranteed timely access to the network. It distinguishes between two basic classes of traffic, synchronous and asynchronous. *Synchronous traffic* is the real-time traffic; the protocol guarantees that each node can send out up to a certain amount of synchronous traffic every T time units. *Asynchronous traffic* is nonreal-time traffic that takes up any bandwidth left unused by the synchronous traffic. It can itself consist of multiple priority classes, but we shall not concern ourselves with that here. We will concentrate solely on the way this protocol handles the synchronous traffic.

The key control parameter of this protocol is the target token-rotation time, TTRT. The protocol attempts to ensure that the cycle time of the token (i.e., the time for the token to make a complete circuit around the nodes of the network) is no more than the TTRT. This is not always possible. However, as we see below, it is possible to guarantee that, barring network failures, the token cycle time is no more than twice the target time ($2 \times$ TTRT). Every time the token visits it, a node is allowed to transmit up to a preassigned quota of synchronous traffic. Thus, if it is necessary to ensure that each node can send out some synchronous traffic once every T units, we can set the TTRT $= T/2$. TTRT can thus be set by interaction between the nodes. Each node indicates the maximum acceptable interval between two successive visits by the token to that node. The minimum of these times is halved, and set as the TTRT, so that the constraints of all the nodes are met. We will return to the choice of the TTRT below.

The total volume of synchronous traffic that can be transmitted during any cycle is easy to calculate. Denote by B the bandwidth of the network in bits per unit time, and by Θ the control overhead per cycle. Then, the time available to carry packets per cycle is given by

$$t_p = \text{TTRT} - \Theta$$

and the total number of bits of traffic that the network can support during a cycle is given by Bt_p. Each node i is allocated a fraction of this quantity, f_i, that it

can transmit during any cycle. That is, it is allocated a quota of $f_i t_p B$ bits of synchronous traffic that it may transmit during any one cycle.

We now have the background to describe the timed-token protocol. When the system begins operating, no data packets are transmitted during the first cycle. Instead, the nodes spend the first cycle determining the value of the TTRT. They do this by broadcasting the value they want (recall that if a node wishes to be able to broadcast at least once every T time units, it needs TTRT $= T/2$), and picking the smallest value requested. During the second cycle of the token, only synchronous traffic is transmitted. The steady-state portion of the algorithm after the end of the second cycle is shown in Figure 6.28. When the token arrives, the node checks to see if the cycle time (i.e., the duration between the current time and the previous arrival time of the token at that node) is greater than the TTRT. If it is, the token is said to be *late*; it transmits only its synchronous traffic (up to the prescribed maximum) and passes on the token to the next node. If it is not (i.e., the token is *early*), it transmits not only the synchronous traffic, but also a certain amount of asynchronous traffic, if it has any awaiting transmission.

How does each node decide how much of its asynchronous traffic to transmit per cycle? There are many ways to do this. The standard way is to allow a packet of asynchronous traffic to begin transmitting so long as the token is not late. That is, if the token was last released by a node n at time t, it will not be allowed to start

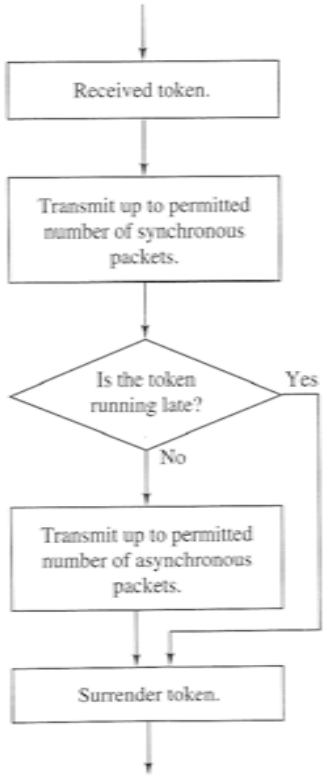

FIGURE 6.28
Flowchart of the timed-token protocol after the second cycle.

transmitting an asynchronous packet if the current time is later than $t +$ TTRT. Note that under this approach, it is possible to have *asynchronous overrun*; that is, it is possible to start transmitting an asynchronous packet just before $t +$ TTRT so that the token is late when the packet is completed. Asynchronous overrun can be regarded as another component of the overhead.

Asynchronous overrun can be easily avoided by not starting an asynchronous packet unless we can ensure that the duration between when the token is given up by this node during the current cycle and when it was given up during the last cycle is no greater than the TTRT. This ensures that the target token-rotation time is not exceeded due to the asynchronous traffic. It is possible to show, although we will not provide the proof here, that the average token-cycle time is no greater than the TTRT.

Protocol analysis. The timed-token protocol is attractive to real-time engineers because it guarantees that the token-cycle time is bounded.

Theorem 6.1. In the absence of failures, the maximum cycle time of the token is no greater than twice the TTRT.

Proof. For ease of exposition, we ignore the impact of the overhead in this proof; incorporating the overhead into the result is quite simple, and is left as an exercise.

Denote by (a, b) the ath visit of the token to node b. That is, it is the visit to node b of the token in the ath cycle. Let $t(a, b)$ denote the time when the token completes its ath visit to node b. The ath token-cycle time, $C(a, b)$ as seen by node b, $C(a, b)$, is defined as $C(a, b) = t(a, b) - t(a - 1, b)$. That is, it is the interval between when the token left node b during its ath visit and when it left node b during its $(a - 1)$st visit. Figure 6.29 illustrates this.

Let $S(x, y)$ and $A(x, y)$ denote the volume of the synchronous and asynchronous traffic, respectively, transmitted during (x, y). Consider visit (a, b) of the token. We now consider three cases.

Case 1. The token has been early or on time throughout the entire cycle preceding visit (a, b). That is, it has not been late during any of the visits

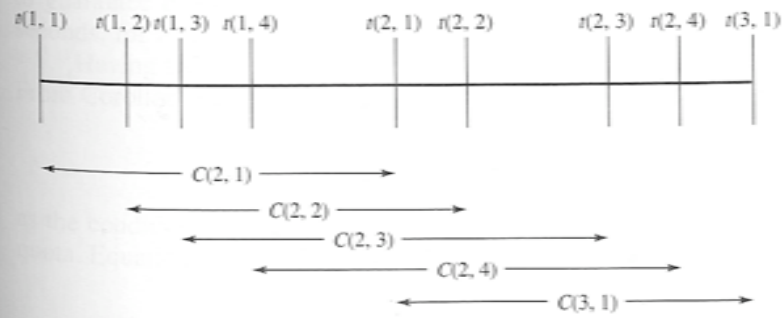

FIGURE 6.29
The ath token-cycle time at node b, $C(a, b)$.

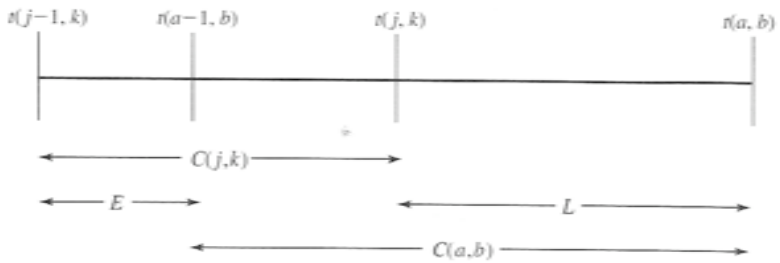

FIGURE 6.30
Case 3: when the token is early for part of the cycle.

$(a-1, b+1), \cdots, (a-1, b-1), (a, b)$.[3] In such a case, there is nothing to prove. The cycle time has, by definition, been less than the TTRT throughout that cycle.

Case 2. The token has been late throughout the entire cycle preceding visit (a, b). In such a case, none of the nodes during this sequence of visits has been allowed to transmit asynchronous traffic; each has transmitted only its quota of synchronous traffic. Since we are ignoring overhead, this means that

$$C(a,b) = \sum_{x,y=a-1,b+1}^{a,b} S(x,y)$$

$$\leq \text{TTRT} \sum_{x,y=a-1,b+1}^{a,b} f_i$$

$$\leq \text{TTRT}$$

indicating that there cannot be two consecutive cycles of duration greater than the TTRT.

Case 3. The token has been early for part of the cycle preceding (a, b). Let (j, k) be the visit preceding (a, b) for which the token was early. See Figure 6.30. Since the token was early at $t(j, k)$, by definition,

$$C(j,k) \leq \text{TTRT}$$

The token is late over the entire interval L, and so only synchronous traffic is broadcast during that time. From Figure 6.30, we have:

$$C(a,b) = C(j,k) - E + L$$

$$\leq \text{TTRT} - E + \sum_{x,y=j,k}^{a,b} S(x,y)$$

$$\leq \text{TTRT} + \sum_{x,y=j,k}^{a,b} S(x,y)$$

[3]All additions are modulo the ring, i.e., if $b = N$, $(a-1, b+1)$ is really $(a, 1)$, where N= number of nodes.

$$\leq \text{TTRT} + \text{TTRT}$$

$$= 2\text{TTRT}$$

This completes the proof. **Q.E.D.**

Theorem 6.2, which we state without proof, generalizes Theorem 6.1.

Theorem 6.2. The total duration of ℓ consecutive cycles of the token is upper-bounded by $(\ell + 1)$TTRT, for $\ell = 1, 2, \ldots$.

Corollary 6.1. It follows from Theorem 6.2 that over any interval of duration τ, node n_i will be able to transmit at least

$$\left\lfloor \frac{\tau}{\text{TTRT}} - 1 \right\rfloor f_i t_p B$$

bits.

What impact does the choice of TTRT have on the available bandwidth? The overhead Θ is essentially a constant per cycle, regardless of the volume of data transmitted.[4] So, if the cycle time is the TTRT, the useful time available per cycle is TTRT $- \Theta$, which means that the utilization of the ring is upper-bounded by

$$\Psi = \frac{\text{TTRT} - \Theta}{\text{TTRT}}$$

We therefore have a tradeoff between the upper bound of the delay, which we have shown to be $2\times$TTRT, and the throughput, which is $\Psi \times B$(the channel bandwidth).

Supporting periodic message loads. The timed-token protocol is well suited to supporting periodic message loads. Suppose that over every period P_i, node n_i has to transmit c_i bits of real-time traffic. Since the token-rotation time is upper-bounded by 2TTRT, the following constraint must be satisfied

$$\text{TTRT} \leq \frac{P_i}{2} \tag{6.4}$$

to guarantee that node n_i will have a chance to transmit at least once every P_i seconds, for all i.

Having fixed the TTRT, our next step is to allocate the synchronous quota. From Corollary 6.1, we immediately have:

$$\left\lfloor \frac{P_i}{\text{TTRT}} - 1 \right\rfloor f_i t_p B \geq c_i \tag{6.5}$$

as the condition necessary to guarantee that node n_i has a sufficient synchronous quota. Equation (6.5) can be solved for f_i. Equations (6.4) and (6.5) are necessary

[4]We assume that asynchronous overrun is prevented from occurring.

and sufficient conditions for node n_i to be able to transmit c_i bits of real-time traffic every P_i seconds.

Who sets the value of f_i? The simple approach is for some central authority to set it to satisfy the necessary and sufficient conditions. However, if c_i changes rapidly with time, this may impose an unacceptable overhead on the system. In such a case, we would like to give each node n_i the freedom to pick its own f_i. The following theorem, stated without proof, provides the answer.

Theorem 6.3. If the system consists of nodes $n_1, \ldots, n_m$, let each node n_i pick the minimum f_i so that the constraint in Equation (6.5) is satisfied for $i = 1, \ldots m$. Let

$$\text{TTRT} = \min\left\{\frac{P_1}{2}, \frac{P_2}{2}, \ldots, \frac{P_m}{2}\right\} \tag{6.6}$$

Then, so long as

$$\sum_{i=1}^{m} \frac{c_i}{P_i} \leq \frac{1 - \Theta/\text{TTRT}}{3} \tag{6.7}$$

every node n_i will be able to transmit c_i bits of synchronous traffic every P_i seconds.

Fault-tolerance. Since a token loss can bring the entire network to a halt, fault-detection and recovery are important in token-based protocols. We have already proved that, under normal operation, the token cannot be late at any station for two consecutive cycles—this is an indication of failure. Ring recovery involves the nodes again negotiating the value of the TTRT; each node i announces TTRT(i), the value of the token-rotation time that it desires, and transmission is restarted with the node that has requested the smallest value. In the event that more than one node has requested the smallest value, the tie is broken by selecting from among them the node with the smallest index. This policy is easy to implement. When a node starts ring recovery, it continuously sends out *claim-token* packets, which contain the value of TTRT requested by that node. When a node receives a claim-token packet, it follows the procedure depicted in Figure 6.31. It is easy to see that only one node will receive its own claim-token packet back; this is the node that will reinitiate transmission. If there is a physical break in the ring or some similar malfunction, no node will receive its claim-token packet back.

Note that every node i should receive either the claim-token packet back or a normal packet within TTRT(i) after it transmitted the claim-token packet. If this does not happen, that is an indication of either another loss of the token, or a loss of the message, or a physical malfunction. To identify physical malfunctions, beacon packets are used, as shown in Figure 6.32. If there is a break in the ring, the only node that will keep transmitting is the station immediately downstream from the break. The system software that controls the ring must then decide how to reconfigure it, by switching in backup links if necessary and available, to restore functioning.

If the deadlines are so short that the delays associated with such recovery techniques are unacceptable, we can use forward error masking. Instead of one physical fiber, we use N and give each processor one physically distinct interface connecting it to each of the fibers. Then we transmit N copies of everything.

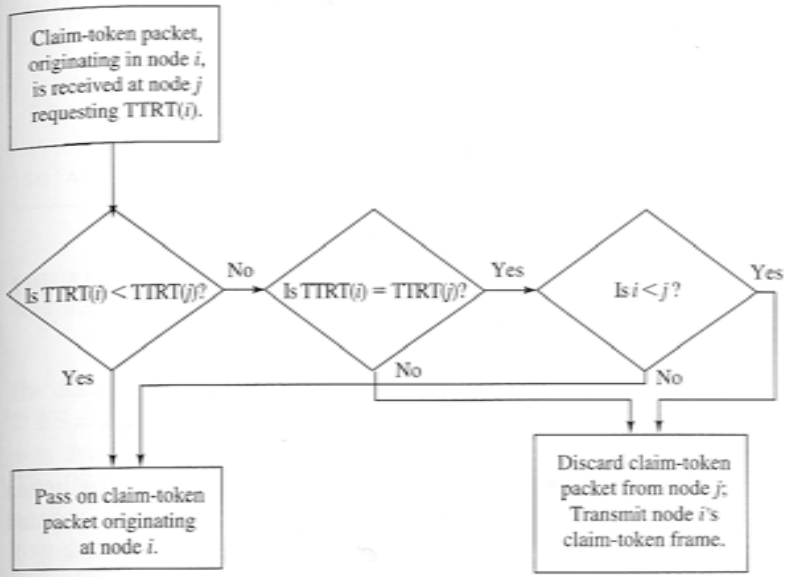

FIGURE 6.31
Handling the claim-token packet.

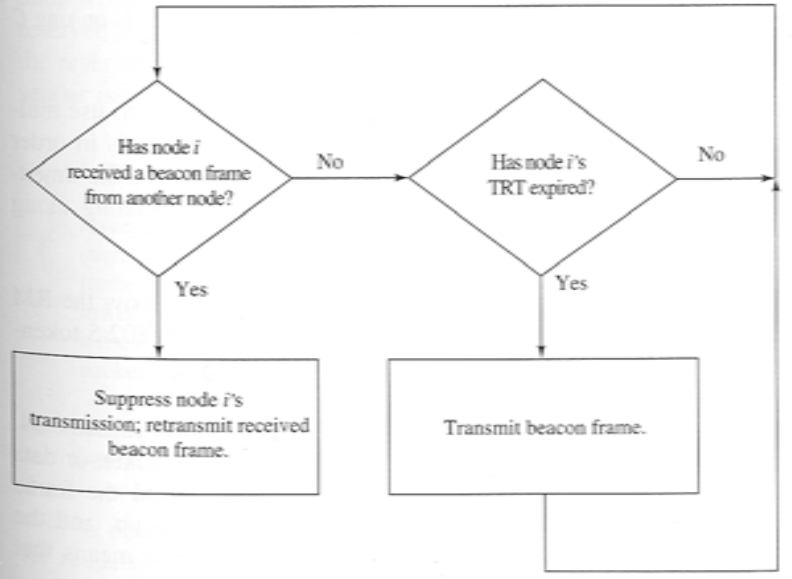

FIGURE 6.32
Beacon packet transmission.

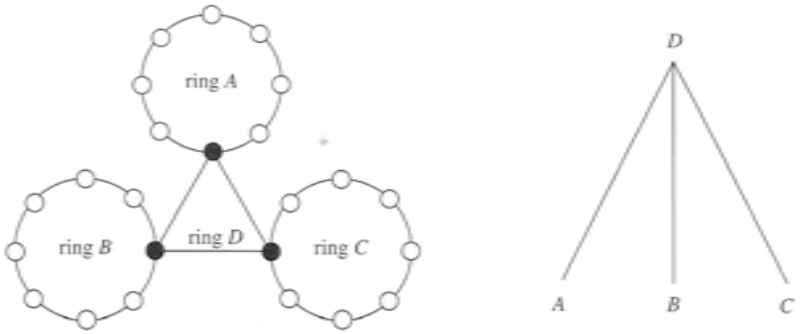

FIGURE 6.33
A two-level structure of rings: (a) ring structure; (b) ring hierarchy.

Use of the timed-token protocol in hierarchical networks. The timed-token protocol can also be used in hierarchical networks, where the nodes are connected in rings and the rings are connected to other rings to form additional levels.

> **Example 6.15.** Figure 6.33 shows a two-level ring structure. Figure 6.33b shows the hierarchy of the four rings. The black nodes in Figure 6.33a are the interconnecting elements, which are responsible for forwarding packets between rings. There is one token circulating through all four rings. A message that originates in ring A and is destined for a node in ring C is sent to the interconnecting element of A, then travels on ring D to the interconnecting element of C, and finally travels on ring C to its destination.

As mentioned earlier in Section 6.1.1, it is frequently necessary to use multiplexing techniques to divide the fiber channel into virtual subchannels in order to better utilize the very large raw fiber bandwidth. It is possible to use the timed-token protocol in such multichannel systems, with one token circulating along each channel.

IEEE 802.5 TOKEN-RING PROTOCOL. In this section, we look at how the RM algorithm (see Chapter 3) can be implemented on the standard IEEE 802.5 token-ring protocol.

Some 802.5 basics. The token and data packet[5] formats are shown in Figure 6.34. The starting and ending delimiters indicate the start and end of the token or data frame. The frame control field indicates that it is a data frame, and the frame status indicates whether or not the destination is present, powered up, and the message has been received successfully. In particular, if FS = 00, it means that

[5]The term *frame* is used synonymously with *packet* in this discussion.

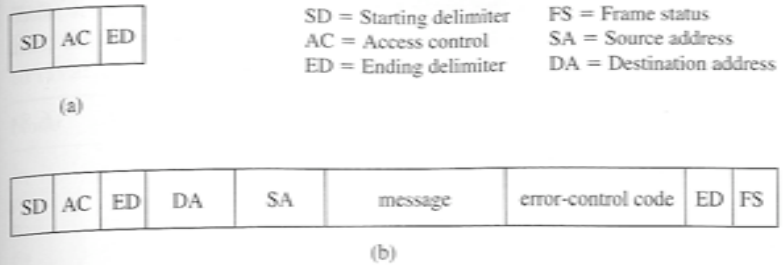

FIGURE 6.34
(a) Token format; (b) data packet format.

the destination node was not available (e.g., it was not powered up or was faulty). If FS = 10, the frame could not copied at the destination, although the destination was available. If FS = 11, the frame was successfully received by its destination. The FS field is checked by the sender when the data frame comes back to it. The sender removes the data frame from the ring when it returns after making one round trip of the ring.

Of particular interest to us is the priority arbitration scheme in this protocol. The access control field contains three bits for the current and reserved priorities. Suppose the highest-priority message at node n_i has priority p_i. When a data frame or token goes by, n_i checks the reserved priority (or reservation) bits. If these indicate a priority greater than or equal to p_i, it does nothing; there is another node wishing to use the network with a message of higher or equal priority. If the reserved priority bits indicate a priority of less than p_i, n_i writes priority p_i onto the reservation field. When the current data transmission has been completed, the sender issues a token with the priority level indicated by the reservation bits.

> **Example 6.16.** Consider a five-node ring, with the highest priorities of the packets awaiting service at the nodes being 2, 4, 1, 6, and 8, at n_1, n_2, n_3, n_4 and n_5, respectively. (The lower the priority index, the greater the priority). The ring is currently serving node n_1. n_1 writes priority 2 in the reservation bits of its data frame and sends it out. n_2 does not change it, since its highest-priority message has lower priority than that. However, node n_3 has a higher priority message, and so writes its priority onto the reservation field. Nodes n_4 and n_5 have lower priority packets, and so they do not update the reservation field.
>
> When the packet returns to node n_1, the node generates a token of priority 1 and sends it out. This token cannot be captured by either n_1 or n_2, since they have lower priority. It is seized by n_3, which then starts transmitting its data packet.

A node that increases the priority of the reservation field is also responsible for reducing it to its prior value. Otherwise, the token priority would never decrease! A node may hold the token for at most a preset token-holding time (the default is 10 ms). Of course, the same node may seize the token multiple times in succession if its message priority equals or exceeds the value written into the reservation field.