6.4.2 Random Early Detection (RED)

A second mechanism, called *random early detection* (RED), is similar to the DECbit scheme in that each router is programmed to monitor its own queue length and, when it detects that congestion is imminent, to notify the source to adjust its congestion window. RED, invented by Sally Floyd and Van Jacobson in the early 1990s, differs from the DECbit scheme in two major ways.

The first is that rather than explicitly sending a congestion notification message to the source, RED is most commonly implemented such that it *implicitly* notifies the source of congestion by dropping one of its packets. The source is, therefore, effectively notified by the subsequent timeout or duplicate ACK. In case you haven't already guessed, RED is designed to be used in conjunction with TCP, which currently detects congestion by means of timeouts (or some other means of detecting packet loss such as duplicate ACKs). As the "early" part of the RED acronym suggests, the gateway drops the packet earlier than it would have to, so as to notify the

source that it should decrease its congestion window sooner than it would normally have. In other words, the router drops a few packets before it has exhausted its buffer space completely, so as to cause the source to slow down, with the hope that this will mean it does not have to drop lots of packets later on. Note that RED could easily be adapted to work with an explicit feedback scheme simply by *marking* a packet instead of *dropping* it, as discussed in the sidebar on Explicit Congestion Notification.

Explicit Congestion Notification (ECN)

While current deployments of RED almost always signal congestion by dropping packets, there has recently been much attention given to whether or not explicit notification is a better strategy. This has led to an effort to standardize ECN for the Internet.

The basic argument is that while dropping a packet certainly acts as a signal of congestion, and is probably the right thing to do for long-lived bulk transfers, doing so hurts applications that are sensitive to the delay or loss of one or more packets. Interactive traffic such as telnet and web browsing are prime examples. Learning of congestion through explicit notification is more appropriate for such applications.

Technically, ECN requires two bits; the proposed standard uses bits 6 and 7 in the IP type of service (TOS) field. One is set by the source to indicate that it is ECN capable; that is, it is able to react to a congestion notification. The other is set by routers along the end-to-end path when congestion is encountered. The latter bit is also echoed back to the source by the destination host. TCP running on the source responds to the ECN bit set in exactly the same way it responds to a dropped packet.

As with any good idea, this recent focus on ECN has caused people to stop and think about other ways in which networks can benefit from an ECN-style exchange of information between hosts at the edge of the networks and routers in the middle of the network, piggybacked on data packets. The general strategy is sometimes called *active queue management*, and recent research seems to indicate that it is particularly valuable to TCP flows that have large delay-bandwidth products. The interested reader can pursue the relevant references given at the end of the chapter.

The second difference between RED and DECbit is in the details of how RED decides when to drop a packet and what packet it decides to drop. To understand the basic idea, consider a simple FIFO queue. Rather than wait for the queue to become completely full and then be forced to drop each arriving packet (the tail drop policy of Section 6.2.1), we could

3.

decide to drop each arriving packet with some *drop probability* whenever the queue length exceeds some *drop level*. This idea is called *early random drop*. The RED algorithm defines the details of how to monitor the queue length and when to drop a packet.

In the following paragraphs, we describe the RED algorithm as originally proposed by Floyd and Jacobson. We note that several modifications have since been proposed both by the inventors and by other researchers; some of these are discussed in Further Reading. However, the key ideas are the same as those presented below, and most current implementations are close to the algorithm that follows.

First, RED computes an average queue length using a weighted running average similar to the one used in the original TCP timeout computation. That is, AvgLen is computed as

$$\text{AvgLen} = (1 - \text{Weight}) \times \text{AvgLen} + \text{Weight} \times \text{SampleLen}$$

where $0 < \text{Weight} < 1$ and SampleLen is the length of the queue when a sample measurement is made. In most software implementations, the queue length is measured every time a new packet arrives at the gateway. In hardware, it might be calculated at some fixed sampling interval.

The reason for using an average queue length rather than an instantaneous one is that it more accurately captures the notion of congestion. Because of the bursty nature of Internet traffic, queues can become full very quickly and then become empty again. If a queue is spending most of its time empty, then it's probably not appropriate to conclude that the router is congested and to tell the hosts to slow down. Thus, the weighted running average calculation tries to detect long-lived congestion, as indicated in the right-hand portion of Figure 6.15, by filtering out short-term changes in the queue length. You can think of the running average as a low-pass filter, where Weight determines the time constant of the filter. The question of how we pick this time constant is discussed below.

Second, RED has two queue length thresholds that trigger certain activity: MinThreshold and MaxThreshold. When a packet arrives at the gateway, RED compares the current AvgLen with these two thresholds, according to the following rules:

$$\text{if AvgLen} \leq \text{MinThreshold}$$

$$\longrightarrow \text{queue the packet}$$

if MinThreshold < AvgLen < MaxThreshold

⟶ calculate probability P

⟶ drop the arriving packet with probability P

if MaxThreshold ≤ AvgLen

⟶ drop the arriving packet

If the average queue length is smaller than the lower threshold, no action is taken, and if the average queue length is larger than the upper threshold, then the packet is always dropped. If the average queue length is between the two thresholds, then the newly arriving packet is dropped with some probability P. This situation is depicted in Figure 6.16. The approximate relationship between P and AvgLen is shown in Figure 6.17. Note that the probability of drop increases slowly when AvgLen is between the two thresholds, reaching MaxP at the upper threshold, at which point it jumps to unity. The rationale behind this is that, if AvgLen reaches the upper threshold, then the gentle approach (dropping a few packets) is not working and drastic measures are called for: dropping all arriving packets. Some research has suggested that a smoother transition from random dropping to complete dropping, rather than the discontinuous approach shown here, may be appropriate.

Although Figure 6.17 shows the probability of drop as a function only of AvgLen, the situation is actually a little more complicated. In fact, P is

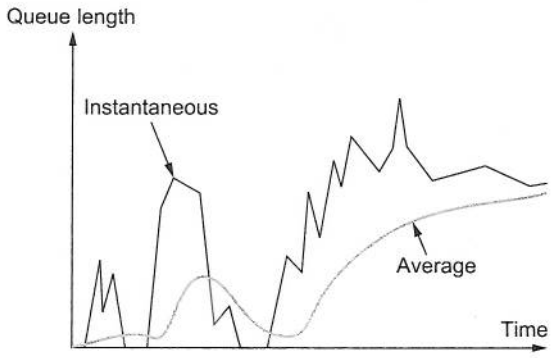
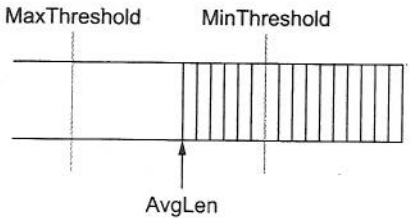
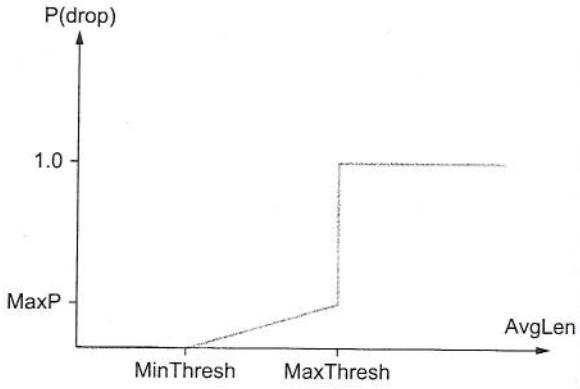


FIGURE 6.15 Weighted running average queue length.



■ **FIGURE 6.16** RED thresholds on a FIFO queue.



■ **FIGURE 6.17** Drop probability function for RED.

a function of both AvgLen and how long it has been since the last packet was dropped. Specifically, it is computed as follows:

$$\text{TempP} = \text{MaxP} \times (\text{AvgLen} - \text{MinThreshold})/(\text{MaxThreshold} - \text{MinThreshold})$$

$$\text{P} = \text{TempP}/(1 - \text{count} \times \text{TempP})$$

TempP is the variable that is plotted on the y-axis in Figure 6.17, count keeps track of how many newly arriving packets have been queued (not dropped), and AvgLen has been between the two thresholds. P increases slowly as count increases, thereby making a drop increasingly likely as the time since the last drop increases. This makes closely spaced drops relatively less likely than widely spaced drops. This extra step in calculating P was introduced by the inventors of RED when they observed that, without

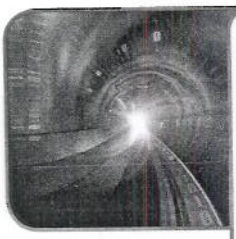
it, the packet drops were not well distributed in time but instead tended to occur in clusters. Because packet arrivals from a certain connection are likely to arrive in bursts, this clustering of drops is likely to cause multiple drops in a single connection. This is not desirable, since only one drop per round-trip time is enough to cause a connection to reduce its window size, whereas multiple drops might send it back into slow start.

As an example, suppose that we set MaxP to 0.02 and count is initialized to zero. If the average queue length were halfway between the two thresholds, then TempP, and the initial value of P, would be half of MaxP, or 0.01. An arriving packet, of course, has a 99 in 100 chance of getting into the queue at this point. With each successive packet that is not dropped, P slowly increases, and by the time 50 packets have arrived without a drop, P would have doubled to 0.02. In the unlikely event that 99 packets arrived without loss, P reaches 1, guaranteeing that the next packet is dropped. The important thing about this part of the algorithm is that it ensures a roughly even distribution of drops over time.

The intent is that, if RED drops a small percentage of packets when AvgLen exceeds MinThreshold, this will cause a few TCP connections to reduce their window sizes, which in turn will reduce the rate at which packets arrive at the router. All going well, AvgLen will then decrease and congestion is avoided. The queue length can be kept short, while throughput remains high since few packets are dropped.

Note that, because RED is operating on a queue length averaged over time, it is possible for the instantaneous queue length to be much longer than AvgLen. In this case, if a packet arrives and there is nowhere to put it, then it will have to be dropped. When this happens, RED is operating in tail drop mode. One of the goals of RED is to prevent tail drop behavior if possible.

The random nature of RED confers an interesting property on the algorithm. Because RED drops packets randomly, the probability that RED decides to drop a particular flow's packet(s) is roughly proportional to the share of the bandwidth that that flow is currently getting at that router. This is because a flow that is sending a relatively large number of packets is providing more candidates for random dropping. Thus, there is some sense of fair resource allocation built into RED, although it is by no means precise.



Note that a fair amount of analysis has gone into setting the various RED parameters—for example, MaxThreshold, MinThreshold, MaxP, and Weight—all in the name of optimizing the power function (throughput-to-delay ratio). The performance of these parameters has also been confirmed through simulation, and the algorithm has been shown not to be overly sensitive to them. It is important to keep in mind, however, that all of this analysis and simulation hinges on a particular characterization of the network workload. The real contribution of RED is a mechanism by which the router can more accurately manage its queue length. Defining precisely what constitutes an optimal queue length depends on the traffic mix and is still a subject of research, with real information now being gathered from operational deployment of RED in the Internet.

Consider the setting of the two thresholds, MinThreshold and MaxThreshold. If the traffic is fairly bursty, then MinThreshold should be sufficiently large to allow the link utilization to be maintained at an acceptably high level. Also, the difference between the two thresholds should be larger than the typical increase in the calculated average queue length in one RTT. Setting MaxThreshold to twice MinThreshold seems to be a reasonable rule of thumb given the traffic mix on today's Internet. In addition, since we expect the average queue length to hover between the two thresholds during periods of high load, there should be enough free buffer space *above* MaxThreshold to absorb the natural bursts that occur in Internet traffic without forcing the router to enter tail drop mode.

We noted above that Weight determines the time constant for the running average low-pass filter, and this gives us a clue as to how we might pick a suitable value for it. Recall that RED is trying to send signals to TCP flows by dropping packets during times of congestion. Suppose that a router drops a packet from some TCP connection and then immediately forwards some more packets from the same connection. When those packets arrive at the receiver, it starts sending duplicate ACKs to the sender. When the sender sees enough duplicate ACKs, it will reduce its window size. So, from the time the router drops a packet until the time when the same router starts to see some relief from the affected connection in terms of a reduced window size, at least one round-trip time must elapse for that connection. There is probably not much point in having the router respond to congestion on time scales much less than the round-trip time of the connections passing through it. As noted previously, 100 ms is not a bad estimate of average round-trip times in the Internet. Thus, Weight should be chosen such that changes in queue length over time scales much less than 100 ms are filtered out.

Since RED works by sending signals to TCP flows to tell them to slow down, you might wonder what would happen if those signals are ignored. This is often called the *unresponsive flow* problem, and it has been a matter of some concern for several years. Unresponsive flows use more than their fair share of network resources and could cause congestive collapse if there were enough of them, just as in the days before TCP congestion control. Some of the techniques described in Section 6.5 can help with this problem by isolating certain classes of traffic from others. There is also the possibility that a variant of RED could drop more heavily from flows that are unresponsive to the initial hints that it sends; this continues to be an area of active research.