Part VI: Advanced Topics (Bonus Material on CD-ROM)

This part includes additional material that are related to Part IV and Part V; it consists of two sub-parts.

In the first sub-part, three chapters (Chapter 21, Chapter 22, and Chapter 23) cover functions and components of a router in further detail as a continuation of Part IV. First, different approaches to architect the switch fabric of a router are presented in Chapter 21. Second, packet queueing and scheduling approaches are discussed along with their strengths and limitations in Chapter 22. Third, traffic conditioning, an important function of a router, especially to meet service level agreements, is presented in Chapter 23.

In the second sub-part, we include two chapters (Chapter 24 and Chapter 25). Transport network routing is presented first in its general framework, followed by a formal treatment of the transport network route engineering problem over multiple time periods, in Chapter 24. The final chapter (Chapter 25) covers two different dimensions: optical network routing and multi-layer network routing. In optical network routing, we discuss both SONET and WDM in a transport network framework; more importantly, we also point out the circumstances under which a WDM on-demand network differs from a basic transport network paradigm. Furthermore, we discuss routing in multiple layers from the service network to multiple views of the transport networks; this is done by appropriately considering the unit of information on which routing decision is made and the time granularity of making such a decision. We conclude by presenting overlay network routing and its relation to multilayer routing.

22 Packet Queueing and Scheduling

We dance round in a ring and suppose,
But the Secret sits in the middle and knows.

Robert Frost

Reading Guideline

In this chapter, we present handling of packets by the queue manager at a router for queueing and scheduling. While this chapter can be read independently, the background on router architectures presented in Chapter 14 is useful in understanding why packet queueing and scheduling is important.

D. Medhi and K. Ramasamy, *Network Routing: Algorithms, Protocols, and Architectures*.
© 2007 by Elsevier, Inc. All rights reserved.

A critical function of a router is to handle queueing of arriving packets and then schedule them on an outgoing interface. This is essentially the job of the "queue manager" at a router, as shown earlier in several figures in Chapter 14. An important requirement is that these functions must be efficient so that a packet leaves a router extremely quickly while giving priority to certain packet types as and when needed. However, in case of heavy traffic, a router might receive many packets almost instantaneously from active microflows, which are then queued in a buffer. Since a buffer is of finite size, it is quite possible that it becomes full—in this case, some packets are to be dropped. The question then is: what rules or policies are to be used for selecting packets to be dropped? The decision on dropping packet is, in fact, dictated to some extent by whether this is desirable from the point of view of an end-to-end delivery to end hosts. Since most microflows going through a router are TCP-based, it is important to understand the basics of TCP congestion control mechanisms to fully comprehend why a particular packet-dropping policy might be better or worse.

In this chapter, we discuss queueing and scheduling issues and algorithms for processing packets, including any priority considerations. In addition, we present an overview of TCP congestion control mechanisms, and a variety of packet-dropping or discarding policies. From a queue manager point of view, when a packet arrives, it is ready to schedule it using one of possible schemes, unless it is to be dropped, for example, due to congestion—and to do this all in an efficient manner.

22.1 Packet Scheduling

There are many possible queueing disciplines for packet scheduling. There are, however, two important issues to consider about a queueing discipline: (1) its performance in terms of bounded delay, and (2) whether it can be implemented efficiently. An additional issue is when there are different traffic classes that require different delay guarantees and, possibly, bandwidth guarantees. In this section, we present several mechanisms and discuss their strengths and weaknesses.

22.1.1 First-In, First-Out Queueing

The idea of first-in, first-out (FIFO) queueing is simple. The first packet that arrives at the router is the first one to be transmitted. Note that FIFO queueing is also referred to as first-come, first-served (FCFS) queueing. FIFO, being the simplest, is easy to implement and presents a low system overhead for software-based routers. The advantage of a FIFO queue is that it provides a predictable delay that packets can experience as they pass through the router. If c is the link speed and B is the maximum buffer size, then the delay bound, D, can be calculated as follows:

$$D \le \frac{B}{c}. \tag{22.1.1}$$

The major limitation of FIFO queueing is its inability to discriminate packets based on service class. For instance, a single flow that has bursty arrival can monopolize the entire buffer space of the queue causing all other flows to be denied service until after the burst is serviced. Thus, this form of queueing is used as the default for an output link in a router in the absence of any other queueing discipline.

22.1.2 Priority Queueing

The basic idea behind priority queueing is to classify the packets for various traffic streams arriving at the input link into one or more priority classes as shown in Figure 22.1. This functionality is needed, for example, for the differentiated services architecture. Traffic streams are also referred as flows, which identify a stream of packets that have certain common parameters and have well-defined priority. For example, common parameters could be the destination IP address block, source IP address, or port number, and a priority could be VoIP packets. Such priority information is then communicated along the path of flow, for example, by marking differentiated services code bits (DSCP) (see Figure 1.3(a) and Figure 22.11); this is typically done closer to the origination of a flow, such as the ingress router. Thus, at an intermediate router, the DSCP bits would serve as the indicator on whether a packet is to be prioritized and the router then associates a queue with each priority class and places the classified packets in the appropriate queues. When choosing a packet to transmit, the router will select the highest priority queue that has a nonempty queue before moving on to the next priority queue. Within each priority class, packets are still scheduled in FIFO order.

The main advantage of priority queueing is the segregation of traffic into different classes, and then one class of traffic is served differently from other classes of traffic. Such segregation allows for the preferential treatment of real-time traffic such as VoIP and interactive video (that are assigned higher priority) over non–real-time traffic such as ftp traffic. A difficulty with the priority queueing discipline is that it does not make any guarantees for a particular priority class. It just allows high-priority packets to cut to the front of the line. Furthermore, since it always processes a higher-priority queue before a lower-priority one, it is possible for a high-priority queue to cause packets in a lower-priority queue to be delayed or dropped if the high-priority queue is receiving a constant stream of packets. This essentially leads to bandwidth starvation for lower-priority traffic. To make it feasible, some form of hard limit needs to be imposed on how much higher-priority traffic is inserted in the queue. Thus, to provide flexibility of choice, priority queueing that operates in one of the two modes, *strict priority queueing* and *rate-controlled priority queueing*, are popular. We discuss them below.

STRICT PRIORITY QUEUEING

This is the mode in which the traffic in higher-priority queues is always scheduled ahead of the traffic in the lower-priority queues—which is discussed above. We know from the above discussion that this could lead to bandwidth starvation for lower-priority traffic when

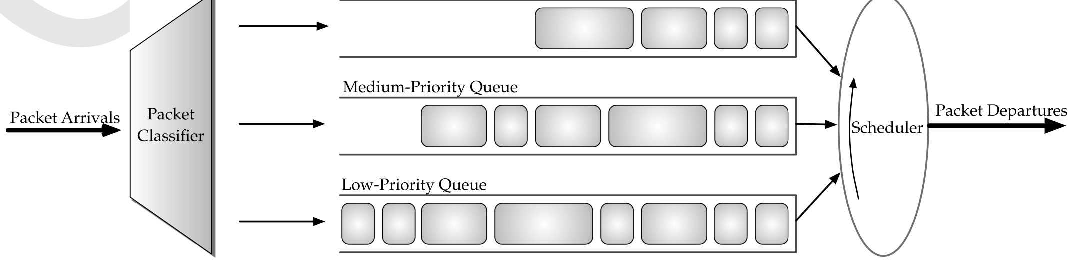


FIGURE 22.1 Priority queueing.

there is an excessive amount of high-priority traffic. However, there are situations in which it is desirable to support strict priority queueing. Consider the scenario in which a service provider, in order to carry VoIP traffic, might have to comply with certain regulations. For example, such a regulation might be that no VoIP traffic should be dropped no matter how congested the network might be. Such a regulation can be supported by using strict priority queueing in which VoIP traffic is placed in a high-priority queue without any limitation on the bandwidth consumed.

Another scenario in which strict priority queueing is used is to protect and prioritize packets that carry routing update information for routing protocols during periods of congestion. Such prioritization is needed to stabilize the route tables when, for example, a topology change occurs.

Rate-Controlled Priority Queueing

Unlike strict priority queueing, rate-controlled priority queueing limits the amount of high-priority traffic so that lower-priority traffic can be scheduled. In other words, rate-controlled priority queueing schedules packets from higher-priority queues before packets from lower-priority queues as long as the amount of traffic in the higher-priority queue stays below a certain threshold.

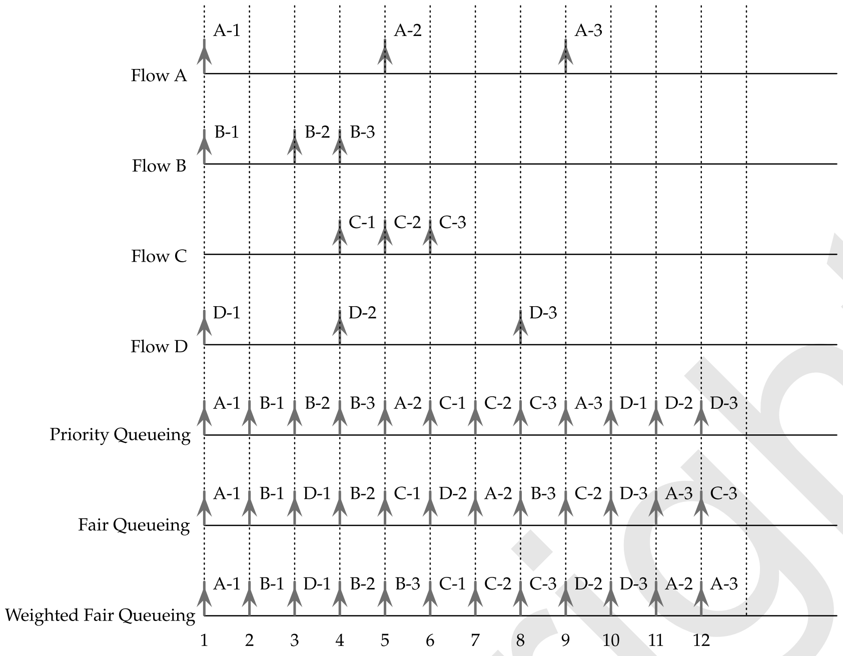
Suppose that a higher-priority queue is rate limited to 25% of the outgoing link bandwidth. As long as the packets from higher-priority traffic consume less than 25% of the output link bandwidth, packets from this queue are scheduled ahead of the lower-priority queues. The moment the higher-priority traffic exceeds 25%, packets in the lower-priority queue can be scheduled ahead of the packets from the higher-priority queue.

22.1.3 Round-Robin and Fair Queueing

Round-robin is similar to priority queueing. It, however, handles multiple traffic classes differently in that it alternates service among different traffic classes (flows). Fair queueing is essentially a round-robin scheme, but can be best described as an approximate bit-by-bit round-robin scheme. Consider Figure 22.1 where we described priority queueing, with three different queues; fair queueing means that the scheduler takes turns in processing a packet from each queue (see also Figure 22.2).

In practice, it is not possible to implement a bit-by-bit scheme for packets arriving for different flows; instead, if there are N active flows, we have to count the clock N bits at a time ("N-bit clock"). Similar to priority queueing, each flow i is assigned a separate queue. If c is the link speed, then each flow i gets a minimum fair share of c/N bits per second on the output. Note that if the link is fully loaded, each queue cannot get more than c/N bits per second. However, if not all flows have packets to send at a particular instant, an individual queue might get higher than c/N for itself.

Suppose that the arriving time of packet k for a flow is A_k, start time for transmission is S_k, and the end time is E_k, respectively. Then, the time needed to send packet k is $E_k - S_k$, to be denoted by packet sending time P_k, i.e., $P_k = E_k - S_k$. We can write this as: $E_k = S_k + P_k$. We, however, need to determine the start time, S_k—this would be arriving time A_k if there is no outstanding packet that is done sending; otherwise, the start time will need to wait until



Flow A
A-1
A-2
A-3
Flow B
B-1
B-2
B-3
Flow C
C-1
C-2
C-3
Flow D
D-1
D-2
D-3
Priority Queueing
A-1 B-1 B-2 B-3 A-2 C-1 C-2 C-3 A-3 D-1 D-2 D-3
Fair Queueing
A-1 B-1 D-1 B-2 C-1 D-2 A-2 B-3 C-2 D-3 A-3 C-3
Weighted Fair Queueing
A-1 B-1 D-1 B-2 B-3 C-1 C-2 C-3 D-2 D-3 A-2 A-3
1 2 3 4 5 6 7 8 9 10 11 12

FIGURE 22.2 Visual comparison of FIFO, FQ, and WFQ scheduler for four traffic classes, from high priority (Flow A) to low priority (Flow D); for illustration, fixed-size packets are shown (adapted from [494]).

the end time of previous packet $(k-1)$ is over. That means the start time for packet k is the maximum of A_k and E_{k-1}. Thus, we can write

$$E_k = \max\{A_k, E_{k-1}\} + P_k, \tag{22.1.2}$$

where A_k, the arrival time of packet k, will be computed using the N-bit clock discussed earlier.

While the fair queueing discipline can guarantee a minimum fair share and bound on the delay for each flow, it faces a different problem from an implementation point of view. As we can see above, for a packet for each flow an end time is calculated—this packet is then added to a queue of packets that are sorted based on the end time. Thus, we face with the problem of inserting a packet into a sorted queue; for N flows, this operation, however, requires $O(\log N)$ time with currently known best algorithms for insertion into a queue. That is, packet processing can be quite expensive if there are too many flows to handle, which is likely, especially in a high-speed core network router.

22.1.4 Weighted Round-Robin and Weighted Fair Queueing

Weighted round-robin (WRR) is also a scheme for handling multiple traffic classes like the round-robin scheme; rather, it is a variation of round-robin in which each traffic class can be assigned a weight. Weighted fair queueing (WFQ) is similarly a variation of fair queueing and can be described as an approximate bit-by-bit *weighted* round-robin scheme. It is not quite apparent where WRR and WFQ are different; to understand their differences, we consider the following example.

Example 22.1 *WRR is fair for packets of fixed length and unfair for packets of variable length.*

Consider three queues Q1, Q2, and Q3 that are serviced by WRR. Assume that the output link bandwidth shared by these queues is as follows. Q1 is allowed to consume 40% of the bandwidth while the remaining 60% of the bandwidth is split equally between Q2 and Q3. If all the packets in all queues are 100 bytes, then at the end of a single service round four packets from Q1 (400 bytes), three packets from queue Q2 (300 bytes), and another three packets from queue Q3 (300 bytes) would have been transmitted. A total of 1000 bytes are transmitted, out of which 400 bytes belong to Q1, thereby receiving its allotted bandwidth of 40%. Similarly, queues Q2 and Q3 transmitted 300 bytes each, thereby receiving their allotted share of 30% each.

Now let us consider the case of packets of variable length, assuming that the WRR scheduler is configured exactly the same as in the previous case for sharing the output port bandwidth. However, the packet sizes are different for each of the queues. Let us assume the mean packet sizes for Q1, Q2, and Q3 are 100 bytes, 150 bytes, and 250 bytes, respectively. After a single round of service, Q1 would have transmitted four packets (400 bytes), Q2 would have transmitted three packets (450 bytes), and Q3 would also have transmitted three packets (750 bytes) for an aggregate total of 1600 bytes. This implies that Q1 received 25% of the bandwidth, which is less than the 40% it is expected to receive. Similarly, Q2 received 28.1%, which is less than the allotted 30%. However, Q3 received 46.9% of the bandwidth, which is greater than its share of 30%. As you can see, the queue that receives a larger mean packet size than other queues consumes more than the configured share of the output port bandwidth. ▲

Note that if packets from different queues have different sizes, the WRR scheduler divides each queue's percentage by its mean packet size to obtain a normalized set of weights. This normalized set of weights indicates how many packets need to be transmitted from each queue in a single round of service. Using the example of packets of variable length, the normalized weights for Q1, Q2, and Q3 are 2/5, 1/5, and 3/25. Converting these weights into whole integers gives 10, 5, and 3 packets.

In general, the rate guarantees in a WFQ scheduler can be described as follows. Consider N flows where each flow i is assigned a weight ω_i. Given the outgoing link capacity to be c, each flow i will receive at least the service rate R_i given by

$$R_i = \left(\frac{\omega_i}{\sum_{n=1}^{N} \omega_n} \right) c. \tag{22.1.3}$$

Since at a particular instant, only a subset of flows would be active, say $\widehat{N}$, we can replace N by $\widehat{N}$ in the above result to obtain the effective fair share. If flow i is allocated a buffer size of B_i, then the delay bound, D_i, for flow i can be given by

$$D_i \leq \frac{B_i}{R_i}. \tag{22.1.4}$$

To summarize, for fixed-size data packets, WRR and WFQ can be thought of as the same and give the same fairness; however, WRR is not as fair for variable-length packets. We can see that WFQ provides nice delay bound and bandwidth guarantee for different streams; however, it suffers from processing complexity since the currently known best algorithms take $O(\log N)$ time for N simultaneous flows [72], [670]. On the other hand, a round-robin approach is efficient from the processing point of view since its processing complexity is constant time, i.e., $O(1)$. Can we combine these ideas? This will be discussed in the next section. Meanwhile, for a visual comparison between FIFO, fair queueing, and WFQ for traffic streams with different priorities, refer to Figure 22.2; for simplicity of illustration, we have used fixed-size packets.

22.1.5 Deficit Round-Robin Queueing

From the above discussion, we can see that often the tussle is between bounding delay and bandwidth guarantee or reducing processing time. Usually, it is favorable to reduce processing time from the viewpoint of a router. However, for many applications, bandwidth guarantee is important—the delay bound is not as critical; furthermore, fairness as in fair queueing (or WFQ) is also desirable. Deficit round-robin (DRR) queueing tries to address this problem [640]. The round-robin approach is desirable from the processing point of view since its processing complexity is constant time, i.e., $O(1)$, as opposed to $O(\log N)$ time for fair queueing. Thus, DRR achieves fairness while having low processing complexity.

DRR can be best illustrated through three key parameters: quantum, Q_i, represents the transmission credits in bytes provided to queue i in the round-robin fashion; deficit counter, C_i^{def}, tracks the difference between the number of bytes that should have been sent and the number of bytes actually sent from queue i in each cycle; and buffer size, B_i, for queue i.

If a larger quantum size is assigned to a queue, it gets a larger share of the bandwidth of the outgoing link. In this scheme, for each queue i, the algorithm maintains a constant Q_i (its *quantum*) and a variable C_i^{def} (its *deficit*).

On each cycle, the algorithm visits each nonempty queue in sequence. For each nonempty queue i, the algorithm will transmit as many packets as possible such that their total size B_i is less than or equal to $Q_i + C_i^{\text{def}}$. If the queue i gets emptied, the algorithm resets C_i^{def} to zero. If it is not reset to zero, C_i^{def} will build up credits indefinitely, eventually leading to unfairness. If the queue is not emptied, then a deficit exists between the number of bytes the algorithm hoped to send $Q_i + C_i^{\text{def}}$ and the number of bytes actually sent B_i. Thus, the deficit of queue C_i^{def} is updated to $Q_i + C_i^{\text{def}} - B_i$ and the algorithm moves on to service the next nonempty queue.

As can be seen from the algorithm, $Q_i + C_i^{\text{def}}$ represents the maximum number of bytes that can be transmitted during a round-robin cycle. Queues that were not able to send as many as $Q_i + C_i^{\text{def}}$ bytes are compensated in the next cycle with $Q_i + C_i^{\text{def}} - B_i$ more bytes to send in addition to the regular quantum Q_i. The updated C_i^{def} represents the bytes to be compensated. The following example will provide a better understanding of the algorithm.

Example 22.2 *Scheduling packets using deficit round-robin.*

Consider the example illustrated in Figure 22.3. The figures show three queues—F_1, F_2, and F_3—consisting of packets that need to be transmitted. For the sake of discussion, assume

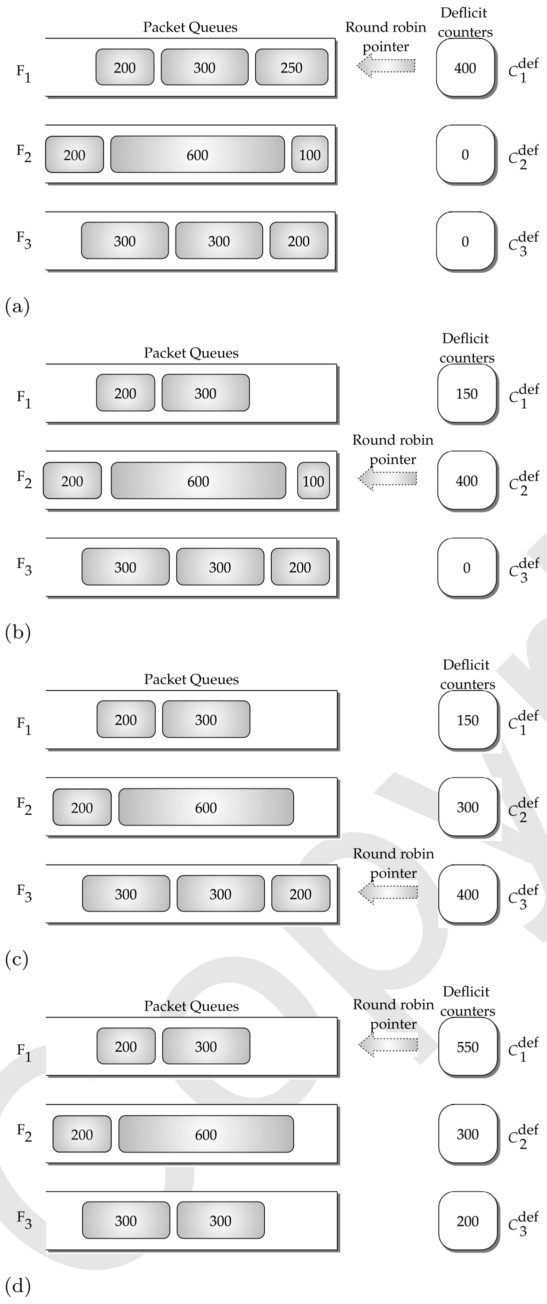


FIGURE 22.3 Illustration of deficit round-robin (DRR).

that the quantum size is 400 bytes. Initially, all C_i^{def} are set to zero and the round-robin pointer points to queue F_1.

The algorithm first allocates a quantum size of 400 to C_1^{def} and as a result it contains 400 transmission credits. This allows queue F_1 to transmit the first packet of size 250 bytes, leaving only 150 bytes, which is not sufficient to transmit the second packet of size 300. Thus, the remaining 150 credits are left as a deficit in C_1^{def}.

Now the algorithm moves to the next non-empty queue F_2. Again, a quantum size of 400 is added to C_2^{def}, which is sufficient to transmit the first packet of size 100 bytes. Thus, the remaining 300 is left as a deficit for the next cycle. Similarly, for queue F_3, the first packet of size 200 is transmitted, leaving a deficit of 200.

The algorithm now returns to the F_1 queue, starting the second cycle. It gives another fresh quantum of 400 credits in addition to the remaining deficit of 150, leaving C_1^{def} at 550. This is sufficient to send the remaining two packets of sizes 300 and 200. The remaining 50 credits could have been saved as a deficit for the packets that arrive later. Instead, it resets C_1^{def} to zero as there are no more packets waiting in the queue and moves to the next nonempty queue F_2.

Again, queue F_2 gets a new quantum of 400 and, thus, the balance becomes 700. This allows the second packet to be transmitted, leaving a deficit of 100. The pointer now moves to F_3, which gets another fresh quantum of 400. Along with a deficit of 200 from the previous cycle, it sends the remaining two packets, leaving a deficit of 0 and no more packets to transmit.

Now the round-robin pointer moves directly to the only nonempty queue F_2. After getting a quantum of 400, F_2 is able to send its third packet of size 200. Since there are no more packets to send, the deficit counter C_2^{def} is zeroed. Hence in three cycles, all the packets have been transmitted. ▲

To ensure that the algorithm serves at least one packet per queue during a cycle, the quantum size needs to be at least $\overline{C}_{\max}$, the MTU size of the outgoing link (alternatively, the largest possible packet size that can be sent).

Irrespective of quantum size, as the size of the packet transmitted can never be larger than the $P_{\max}$ of the outgoing link, C_i^{def}, at any instant, can never be greater than $\overline{C}_{\max}$. If the algorithm has cycled all the queues, say N, times, then the expected number of bytes sent by queue i is $N \times Q_i$. If the actual bytes sent is B_i, then $N \times Q_i - B_i$ will be less than $\overline{C}_{\max}$.

Assigning different quantum values to each queue leads to an allocation of a different percentage of the outgoing link bandwidth for the corresponding queues. If Q_i (quantum size of queue i) is twice the value of Q_j (quantum size of queue j), then queue i will receive twice the bandwidth of j when both queues are active.

The main issue in implementing DRR is eliminating the examination of empty queues. If the number of empty queues is much larger than the number of active queues, a substantial amount of time spent is wasteful. To eliminate this waste, the algorithm maintains a separate queue called *ActiveList*. The *ActiveList* contains a list of the queue indices that contain at least one packet. Each entry in the *ActiveList*, in addition to the queue index, keeps track of its unused deficit as well.

Referring to the example, the *ActiveList* will contain the indices for all the three queues (F_1, F_2, and F_3) with F_1's being at the head. After servicing queue F_1, it will be placed at the tail of the *ActiveList*. If the serviced queue becomes empty (no packets to transmit), then the queue is removed from the *ActiveList*. The use of *ActiveList* provides the advantages of empty queues not being examined and further prevents an empty queue getting from a quantum added when it is idle.

22.1.6 Modified Deficit Round-Robin Queueing

Modified deficit round-robin (MDRR) is a variation of DRR that also addresses delay minimization for some traffic streams. This is helpful, for example, in handling VoIP packets. Although VoIP streams need both bandwidth and delay guarantee, we can think of an alternate approach of providing bandwidth guarantee with minimizing delay. The MDRR scheme then addresses how to minimize delay on top of the DRR scheme. For example, such a modification can be done by assigning priority to queues, which then serve as scheduling priority for different queues. For example, an ultra–high-priority queue may be defined, which always gets priority if it has packets to send and is not restricted by the quantum size. It is important to note that *modification* as in MDRR does not necessarily mean that this is a unique approach; different modifications can be employed; for example, see [379]. It is worthwhile to note that most router vendors currently implement some form of MDRR queueing.

22.2 TCP Congestion Control

In this section, we present an overview of TCP congestion control. Why do we need to know about TCP congestion control in regard to packet queueing and scheduling? We first address this point before presenting TCP congestion control mechanisms. Currently, most commonly used applications in the Internet are TCP-based. To provide good throughput, yet be conscious about congestion, is the basic philosophy of TCP congestion control mechanisms. From a router point of view, it is important to see what mechanisms it must have so that it is TCP-friendly; second, a router should be fair to different TCP microflows in the presence of congestion as well. It is important to understand the distinction between fairness in TCP congestion control as opposed to fairness in packet scheduling. Packet scheduling handles fairness once it has already been decided that a packet is to be processed for queueing; however, fairness as in TCP congestion control refers to how a buffer handles dropping of packets (*before* scheduling) for different TCP microflows so that one or more microflows do not unduly suffer. Thus, we start with TCP congestion control in order to understand benefits or drawbacks of different packet-dropping mechanisms at a router.

When one or several TCP connections send packets at high rates, the network can suffer from congestion. At these high rates, intermediate routers receive more packets than they can buffer. As a result, some packets or their corresponding ACKs may experience more delay or may be dropped before reaching their destination. These dropped packets would trigger timeout at their sources, resulting in retransmission. Such retransmissions increase the number of packets entering the network, thus worsening congestion by introducing more delay and packet drops. These conditions continue until all the resources in the network are utilized to carry packets part of the way before they are dropped and essentially the network is not doing any useful work. Such a condition is called *congestion collapse*.

To avoid congestion collapse from occurring, new mechanisms were introduced into TCP [334]. These mechanisms are based on the principle of conservation of packets. The idea is that a new packet is not injected into the network until an old packet leaves. The sender uses the reception of an ACK as an indication that the packet sent earlier has left the network and initiates the next packet transmission without adding to the level of congestion. This implies that the data packets leave the sender at the same pace as the ACKs arrive at the sender and, thus, TCP is said to be *self-clocking*.

In addition, the congestion control mechanism determines the available capacity in the network so that the sender knows how many packets it can safely have in transit. This property can be used to limit the number of transit packets in the network.

For each TCP connection, the congestion control mechanism tracks several parameters such as *congestion window*. The congestion window, denoted CongWindow, indicates how many bytes the source is allowed to have in transit at any given time. The congestion window is not more than the advertised window (AdvWindow), advertised by the received, that is used for flow control. Using both the congestion window and advertised window, the sender calculates the effective window (EffWindow), which represents the amount of unacknowledged data that a source can have in transit at a given time. Specifically, the effective window may not exceed the minimum of the congestion window and advertised window, i.e.,

$$\begin{aligned} \text{MaxWindow} &\leq \min\{\text{CongWindow}, \text{AdvWindow}\} \times \text{EffWindow} \\ &= \text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked}). \end{aligned} \tag{22.2.1}$$

This equation implies that the source is not allowed to send at a rate faster than what can be handled either by the network or the destination. For example, consider that CongWindow is 10 Kbytes and AdvWindow is 1 Kbytes; if the receiver of a TCP connection is able to accept traffic at 1 Kbytes (AdvWindow of 1 Kbytes) and the network is capable of delivering traffic at 10 Kbytes (CongWindow of 10 Kbytes), then the sender uses the minimum of AdvWindow and CongWindow, which is 1 Kbytes. Alternatively, if the receiver is able to accept traffic at 5 Kbytes and the network is capable of handling 1 Kbytes at that time, then the sender will send at 1 Kbytes speed, even if the receiver can handle 5 Kbytes, for congestion not to occur. The threshold variable determines how CongWindow can grow.

However, how does TCP know the right value for the congestion window? While the value for AdvWindow is explicitly sent by the receiver during TCP connection setup, there is no entity to send a suitable value for CongWindow to the sender. Instead, the sender estimates the level of congestion in the network and accordingly sets a value. Depending on the level of congestion, the value for congestion window is varied, which is decreased when the level of congestion goes up and increased when the level of congestion goes down.

Now the obvious question is how does the source detect congestion? We know that a lost packet triggers TCP timeouts. Such a packet loss can occur due to noise in the transmission line or as a result of congestion in the intermediate routers. With recent advances in transmission technologies, it is now relatively rare that the packet is dropped because of an error during transmission. Thus, TCP assumes that timeouts are caused by congestion and reduces its rate of transmission.

22.2.1 Slow Start

TCP uses a three-way handbook to establish a connection. Once a TCP connection is established between the two end systems, the application process at the sender writes bytes to the sender's TCP send buffer. TCP grabs chunks, which can be at most the size of the maximum segment size (MSS), encapsulates each chunk within a TCP segment, and passes the segments to the network layer for transmission across the network. The TCP congestion window regulates the times at which the segments are sent into the network (i.e., passed to the network layer).

Initially, CongWindow is set equal to one packet. It then sends the first packet into the network and waits for an acknowledgment. If the acknowledgment for this packet arrives before the timer runs out, the sender increases CongWindow by one packet and sends out two packets. Once both these packets are acknowledged before their timeouts, CongWindow is increased again by two. Now the size of CongWindow is four packets and, thus, the sender transmits four packets. Such an exponential increase continues as long as the size of CongWindow is below a defined slow start threshold and acknowledgments are received before their corresponding timeouts expire. After reaching the threshold, the increase is linear until MaxWindow size is reached.

We can see that during this phase the size of the congestion window increases exponentially, i.e., the congestion window is initialized to one packet; after one RTT the window size is increased to two packets; after two RTTs the window is increased to four packets; after three RTTs the window is increased to eight packets; and so on. This mechanism is referred to as *slow start*. If slow start increases the congestion window exponentially, why it is then called slow start? Before slow start was developed, the sender, after establishing the connection, immediately starts transmitting the entire AdvWindow worth of packets at once. While there could be enough bandwidth in the network, the intermediate routers might not have enough buffers available to absorb this burst of packets. Thus, slow start was introduced so that the packets are spaced enough to avoid this burst. In other words, the slow start is considered much slower than transmitting an entire AdvWindow of packets all at once.

The slow start phase ends when the size of CongWindow exceeds the value of a defined congestion threshold. Once the congestion window is larger than the current value of the threshold, the congestion window grows linearly rather than exponentially. Specifically, if W is the current value of the congestion window, and W is larger than threshold, then after W acknowledgments have arrived, TCP replaces W with $W+1$. This has the effect of increasing the congestion window by one in each RTT for which an entire window's worth of acknowledgments arrives. This phase of the algorithm is called *congestion avoidance*.

The congestion avoidance phase continues as long as the acknowledgments arrive before their corresponding timeouts. But the window size, and, thus, the rate at which the TCP sender can send, cannot increase forever. Eventually, the TCP rate will be such that one of the links along the path becomes saturated, and at which point, loss (and a resulting timeout at the sender) will occur. When a timeout occurs, the value of threshold is set to half the value of the current congestion window, and the congestion window is reset to one MSS. The sender again grows the congestion window exponentially fast using the slow start procedure until the congestion window hits the threshold.

22.2.2 Additive Increase, Multiplicative Decrease

Suppose the end-to-end path between two hosts is congestion-free; we still then have the question of how to adjust window size in order to regulate the rate at which the end hosts transmit. TCP uses a principle called *additive increase, multiplicative decrease* (AIMD) for rate adjustment. This means that when increasing the rate, TCP uses an additive property, and when decreasing the rate, it uses a multiplicative property. To be more specific, TCP rate regulation is governed by the parameter, CongWindow; this parameter is increased in additive chucks to increase the rate. Suppose the current CongWindow is 500 bytes and the increase size is 30 bytes, then CongWindow will become 530 bytes. The additive property is less aggressive and is helpful in congestion avoidance. However, when there is congestion, which is indicated by loss of packets, TCP reduces the window size by half. Thus, if we are currently at effective window size of 530 bytes, then it becomes 265 bytes. Note that AIMD is a general principle; TCP window size must still remain valid such as being not higher than AdvWindow. Thus, several boundary conditions are checked; in fact, in actual implementation, the code is intertwined with slow start, congestion avoidance, and fast retransmit and fast recovery, which will be discussed in the next section. For detailed illustration of various parameters, and the relation between TCP segment size, maximum segment size, and the windowing parameters maintained in byte count, see [668].

22.2.3 Fast Retransmit and Fast Recovery

The algorithms described so far, AIMD and slow start, are considered the main TCP congestion control mechanisms. In this section, we consider relatively newer features.

In early implementations of TCP, the sender retransmitted an unacknowledged packet only after the expiration of its timeout. It was observed that the coarse-grained implementation of TCP timers led to long periods of time during which the connection went dead. Thus, a new mechanism called *fast retransmit* was introduced that allows the retransmission of a lost packet even if the timeout for that packet has not expired. The fast retransmit mechanism does not replace the timeout mechanism of TCP, but it works in parallel to improve performance.

The idea behind fast retransmit is straightforward. Every time a packet arrives out of order because the previous packet was lost or delayed, the receiver sends an acknowledgment that is the same as the one sent the last time. The subsequent transmissions of the same acknowledgment are called *duplicate ACKs*. When the sender detects a duplicate ACK, it knows that the receiver must have received a packet out of order, implying that the earlier packet was lost or delayed. To detect reliably the packets that are lost, the sender waits until it sees some number of duplicate ACKs before retransmitting the missing packet. In practice, the sender waits until it has seen three duplicate ACKs, then retransmits the packet without waiting for its timer to expire.

Finally, there is another improvement we can make. Using the fast retransmit mechanism the sender detects a possible loss of a transmitted packet, implying congestion, and therefore it is necessary to reduce its congestion window accordingly, after the transmission of the lost packet. However, when a fast retransmit algorithm is used and when duplicate ACKs are received by the sender, it means that the packets are still flowing to the receiver. How can it be safely deduced? The generation of duplicate ACKs at the receiver is triggered only on a packet arrival. This indicates that serious network congestion may not exist and the lost packet could be a transient condition. Thus, the sender, instead of reducing its transmission rate sharply using slow start, decreases the congestion window by half and performs only additive increase.

22.3 Implicit Feedback Schemes

With the above background on TCP congestion control, we are now ready to discuss *router* congestion control schemes. In general, router congestion control schemes can be classified as implicit feedback schemes and explicit feedback schemes. The main difference in these approaches is that implicit schemes trigger packet dropping while explicit schemes are notifications generated by routers to end hosts to "slow down."

Before we discuss implicit congestion schemes, we need to ask the following question: is packet dropping required because buffers are not sized properly? Recall that we discussed earlier about buffer sizing from a traffic engineering point of view in Section 7.2.3. However, congestions do occur in live networks due to sudden increase in traffic; thus, a buffer sized for normal circumstances might not be adequate for congested situations. This could mean that buffer sizes should be increased further to allow for sudden bursts in traffic; however, there is a downside to having too large of a buffer than required—it can unduly hold packets in a buffer, thus increasing overall latency. That is, it is important to size buffers properly and allow packet dropping intelligently for congested situations.

In implicit feedback schemes, the source detects the existence of congestion by making local observations. Some of these local observations include timeouts when acknowledgments are not received, delayed reception of acknowledgments, arrival of duplicate acknowledgements, and so on. The main underlying causes for these observations are packet delay and packet loss. If the delay experienced by a packet is higher than expected from a source's perspective, it is as good as a packet being lost. As a result, in these schemes the routers simply drop packets during congestion and expect the source to respond to these lost packets by slowing their transmission rate.

22.3.1 Drop Position

This design choice pertains to the position in the queue from which the packet needs to be dropped. For a queue, there are three choices; front of the queue, tail of the queue, and any one of the intermediate positions between them. We examine these choices in detail and identify its advantages and disadvantages.

DROP FROM FRONT

In this case, the space for the arriving packet is created by discarding the packet currently at the head of the queue, as shown in Figure 22.4. While it might sound simple, drop from front is complex to implement. It requires explicit queue manipulation to remove an existing entry, thus requiring extra cycles of processing time. However, the main benefit of dropping from the front of queue is that it expedites TCP's congestion avoidance behavior. That is, drop from front causes the destination to "see" missing packets in its stream earlier.

In fact, drop from front enables a congestion signal being expedited by as much as one RTT time earlier than would be the case with tail drop (where packets are dropped from the tail of the queue). As a result, a few sources whose packets are near the front at the onset of congestion receive earlier congestion notification (that is, they timeout earlier). The ensuing reduction of transmit rate from these sources allows packets from other sources to successfully enter and leave the queue. Thus, fewer sources losing packets greatly reduces or eliminates later overreaction by more sources.

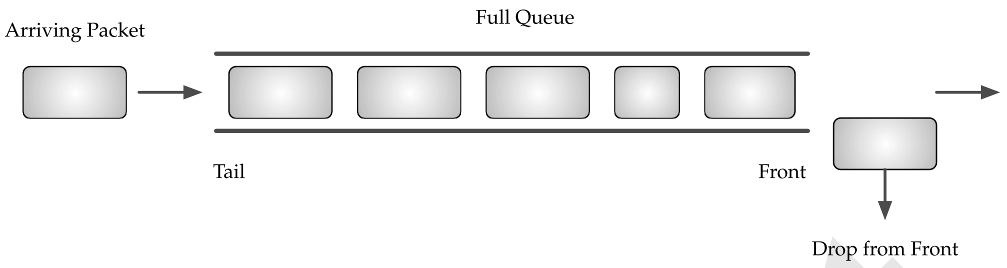


FIGURE 22.4 Drop from front.

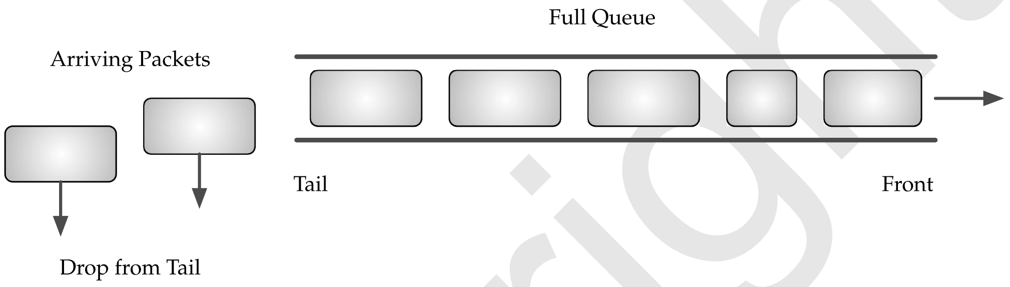


FIGURE 22.5 Drop from tail.

You might wonder why the source timeouts as much as one RTT earlier. This is because of router buffers being sized to hold one RTT worth of traffic. Assume, for the sake of discussion, there are two packets from the same source; one is at the front of the queue while the other is at the end of queue. In the worst case, the packet at the end will be dropped; after all, the packets in the router buffer have been serviced which would consume as much as one RTT. Therefore, if the packet at the front of the queue is dropped, the source would have been notified of congestion one RTT earlier.

TAIL DROP

Another discipline for managing queue length in a router is to determine the maximum length for each queue *a priori*, accept packets until the queue is full, and then discard subsequent packets arriving at the tail of the queue. The dropping behavior continues until space is available in the queue, when packets at the head of the queue are transmitted. This discipline is known as *drop from tail*, or simply tail drop, as the packet that arrived most recently gets discarded when the queue is filled up; this is illustrated in Figure 22.5. The major advantage of tail drop is that it is easy to implement as packets can be dropped simply by not inserting them into the queue. However, it suffers from a few major disadvantages [88].

In some situations, a few flows sharing the output link can monopolize the queue during periods of congestion. Such flows generate packets at a higher rate that can easily fill up the queue. Consequently, the packets from flows generating packets at substantially lower rates have a higher probability of arriving at the queue when it is full and gets discarded. Hence, these lower rate flows are said to be "locked out" by a few higher rate flows.

In a TCP flow, packets often arrive in bursts despite constraint by a window size. With tail drop, when such bursts from several TCP flows arrive at a queue that is full or almost full, their packets will be dropped. As a result, all the flows throttle back by decreasing their transmit window size almost simultaneously. Now all the senders steadily increase their transmission window size in tandem. As a result, the sender sends more packets and again the router queue will overflow, leading to dropping of more packets. This recurring pattern where many senders reduce and increase their transmission rates at the same time is referred as *global synchronization*. Such a phenomenon causes drastic oscillations in traffic, resulting in low utilization of the link bandwidth and of course, reducing the overall network throughput.

Since tail drop signals congestion only when the queue is full, the queues tend to remain full or nearly full for sustained periods of time. As a result, the packets at the end of the queue wait longer before transmission, which increases the latency of flows. Simply making the queue shorter will decrease the latency, but it defeats the purpose of buffering, which is to accommodate brief bursts of traffic without dropping packets unnecessarily. Because of such disadvantages, tail drop is used as the default mechanism for congestion control in the router, if nothing is configured.

22.3.2 Proactive versus Reactive Dropping

This design choice concerns when a packet should be dropped. If packets are dropped even before the queue is full, it is referred as *proactive dropping*. Such proactive dropping of packets implicitly conveys to the end hosts when congestion is imminent. This early notification is useful in networks where the end hosts react by reducing their transmission rate. On the other hand, if the end hosts do not reduce their rate, the router queues will become full and the packets will be dropped anyway. This is called *reactive dropping* as the packets are dropped in reaction to the queue buffers being filled.

For aggregated queues, two variants of proactive dropping have been outlined in the literature: *early random drop* [284] and *random early detection* [227]. In early random drop, when the queue length exceeds a preconfigured threshold, the router drops each arriving packet with a fixed drop probability. Intuitively, as aggressive sources send more packets, dropping an arriving packet randomly will quite likely discard packets from these sources than a packet from one of well-behaved sources. Thus, this scheme attempts to protect the bandwidth received by the well-behaved sources while penalizing the aggressive ones. However, it has been shown in [762] that this scheme was not successful in controlling misbehaving users. Some of the shortcomings of early random drop are improved in random early detection discussed in the next section.

The proactive dropping of packets is called *active queue management* (AQM) in the literature. By dropping packets before buffers overflow, AQM allows routers to control when and how many packets to drop. Active queue management provides the following benefits:

- First, it reduces the number of packets dropped in the router. As the Internet traffic is bursty, a router might not be able to absorb these bursts if most of the queue space is committed to steady-state traffic. By keeping the average queue length small, AQM provides the capability to absorb traffic bursts without discarding packets.

- Second, it eliminates the global TCP synchronization that results in more efficient utilization of network. Second, by keeping the average queue size small, AQM controls the queueing delay experienced by the packets flowing through it. This is significant for interactive applications as their performance largely depends on minimizing the end-to-end delay. Finally, it prevents the aforementioned lockout behavior by ensuring that there is buffer space available for an incoming packet most of the time.

22.4 Random Early Detection (RED)

The basic idea behind random early detection (RED) [227] is to detect incipient congestion *early* and convey congestion notification to the end-hosts, allowing them to reduce their transmission rates before queues in the network overflow and packets are dropped. A router implementing RED continuously monitors the average queue length; when this exceeds a threshold, it randomly drops arriving packets with a certain probability, even though there may be space to buffer the packet. The dropping of a packet serves as an early notification to the source to reduce its transmission rates.

The RED algorithm uses the exponential weighted moving average approach (see Appendix B.6) to calculate average queue length Q_{avg} and to determine when to drop packets. The average queue length is compared with two queue length thresholds, a *minimim* threshold Q_{min} and a *maximum* threshold, triggering certain activity. When a packet arrives at the queue, the RED algorithm compares the current average queue length Q_{avg} with these two thresholds, Q_{min} and Q_{max} (Figure 22.6) according to the following rules:

- If average queue length Q_{avg} is less than the minimum threshold, Q_{min}, no drop is taken and the packet is simply enqueued.
- If average queue length Q_{avg} is greater than the minimum threshold, Q_{min}, but less than the maximum threshold, Q_{max}, it indicates some congestion has begun and the packet is dropped with some probability P_a.
- If average queue length Q_{avg} is greater than the maximum threshold, Q_{max}, it indicates persistent congestion and the packet is dropped to avoid a persistently full queue.

The probability P_a is a function of average queue length Q_{avg} and is often referred to as the *drop probability*. As shown in Figure 22.7, the drop probability is zero when average queue length Q_{avg} is less than or equal to Q_{min}. It increases linearly when Q_{avg} is between

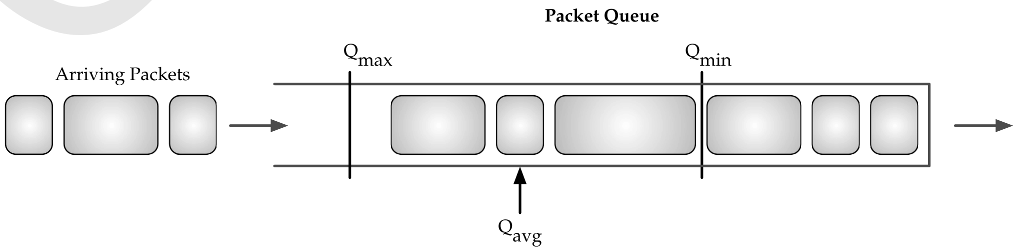


FIGURE 22.6 RED thresholds on a queue.

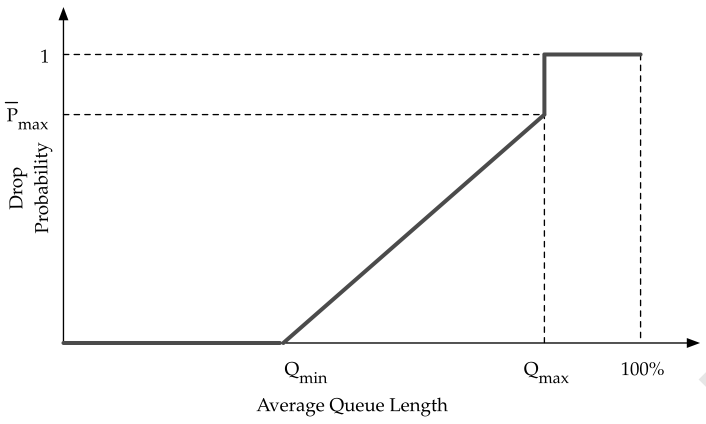


FIGURE 22.7 Illustration of the relationship among Q_{min}, Q_{max}, Q_{avg}, and $\overline{P}_{max}$.

the thresholds Q_{min} and Q_{max}. When Q_{avg} equals Q_{max}, the drop probability reaches P_{max}, at which point it jumps to unity. This indicates that the gentler approach of probabilistically dropping packets is not effective, and aggressive measures need to be taken, that is, dropping all arriving packets.

By using a weighted average, RED avoids overreaction to bursts and instead reacts to longer-term trends. The average queue length captures the notion of congestion more accurately than the instantaneous queue length. The bursty nature of Internet traffic can fill up a queue quickly for a very short period of time, which then becomes empty again. Thus, it is not appropriate to conclude that the router is congested. As a result, the computation of average queue length uses a weighted running average w to detect persistent congestion by filtering short-term variations in the queue length.

While RED is in operation, it is definitely possible that the instantaneous queue length can be much longer than the average queue length Q_{avg}, especially in the presence of bursty traffic. In such situations, when a packet arrives to the router and if the queue is full, then it will be dropped. When this happens, RED is operating in tail drop mode.

An interesting aspect of RED is that it provides some sense of fair resource allocation among flows, due to its random nature. However, the fairness might not be guaranteed to be precise. Because RED drops packets randomly, the probability that a packet is dropped from a particular flow is roughly proportional to the share of bandwidth the flow is getting at that router. Since high-bandwidth flows send a large number of packets through the router, it is providing more candidates for random dropping, thus penalizing them in proportion.

The four parameters that govern the operation and behavior of RED—minimum threshold Q_{min}, maximum threshold Q_{max}, drop probability P_{max}, and weight α used by exponential weighted average—constitute a RED *drop* profile. Realizing RED functionality in a router requires the implementation of two algorithms. The first algorithm computes the average queue length on every packet arrival, while the second algorithm calculates the drop probability that determines the frequency of packets dropped by the router, given the level of congestion. The following sections examine these in detail.

22.4.1 Computing Average Length of Queue

The average queue length, Q_{avg}, is computed using an exponential weighted moving average (refer to Appendix B.6) as

$$Q_{\text{avg}} = (1-w) \times Q_{\text{avg}} + w \times Q_{\text{sample}} \tag{22.4.1}$$

where $0 \le w \le 1$. Q_{sample} represents the actual length of the queue at the instant the measurement is made. In most software implementations, Q_{sample} is measured every time a packet arrives at the router. In hardware, due to high speed requirements, it is calculated at some fixed sampling interval.

Looking at Eq. (22.4.1) more closely reveals that if w is small, even if Q_{sample} is large, Q_{avg} will only increase by a small amount. As a result, Q_{avg} will increase slowly and a significant number of samples of Q_{sample} will be required to increase it substantially. This leads to the detection of long-lived congestion rather than short-term congestion that can come and go.

If w is too small, then Q_{avg} responds too slowly to changes in the actual queue length and is unable to detect the initial stages of congestion. Alternatively, if w is too large, the average queue length will not filter out short-term congestion. Thus, the choice of an appropriate value for w depends on Q_{min} and the amount of burstiness desired. Given a minimum threshold Q_{min}, and the desired level of burstiness as L packets, then w should be chosen to satisfy the following equation [227]:

$$L + 1 + \frac{(1-w)^{(L+1)} - 1}{w} < Q_{\text{min}}. \tag{22.4.2}$$

The left term of the inequality represents the average queue length after L packet arrivals, assuming the queue is initially empty with an average queue length of zero and the queue length increases from 0 to L packets. The inequality implies that if w is chosen appropriately, the router can accept a burst of up to L packets and still manage to keep Q_{avg} below the minimum threshold Q_{min}.

Recall that RED drops packets to signal congestion to TCP flows. Consider a router, dropping a packet from a TCP connection and immediately forwarding subsequent packets from the same connection. When these packets arrive at the destination, it sends duplicate ACKs to the sender. When the sender sees these duplicate ACKs, it reduces its window size. Thus, the time elapsed between the router dropping a packet from a connection and the same router seeing some reduced traffic from the affected connection should be at least one RTT. Practically speaking, there is not much return in having the router respond to congestion.

22.4.2 Computing Drop Probability

A straightforward approach for computing the packet drop probability uses a linear function of the average queue length as shown below:

$$P_a = \overline{P}_{\text{max}} \, (Q_{\text{avg}} - Q_{\text{min}})/(Q_{\text{max}} - Q_{\text{min}}). \tag{22.4.3}$$

In this approach, as the average queue length increases, P_a increases proportionally and reaches a maximum allowable value, $\overline{P}_{\text{max}}$, when the average queue length reaches the maximum threshold Q_{max}. Note that $\overline{P}_{\text{max}}$ is a configurable value in the range, $0 \le \overline{P}_{\text{max}} \le 1$. Even though it is simple to understand and implement, use of this approach leads to dropping packets that will not be well distributed in time. Instead, it is likely to drop more than one packet in a burst of closely spaced packets (clusters) from a source. As the packets of a flow tend to arrive in bursts, such a behavior is likely to cause multiple drops in a single flow. While a single drop per RTT will suffice to cause a flow to reduce its transmit window size, the desirable behavior is to affect many flows so that they can reduce the rate of transmission, thereby mitigating the congestion or reducing the likelihood of congestion occurring immediately.

To reduce the likelihood of such scenarios, the calculation of packet-dropping probability takes into account the number of packets queued since the last drop and that the packet is marked proportional to its size compared to the maximum packet size, MaxPacketSize. This enhanced approach uses the following adjusments for computing the drop probability P_a.

$$P_b = \overline{P}_{\max}(Q_{\text{avg}} - Q_{\min})/(Q_{\max} - Q_{\min}) \tag{22.4.4a}$$

$$P_b = P_b \times \text{ PacketSize/MaxPacketSize} \tag{22.4.4b}$$

$$P_a = P_b/(1 - count \times P_b). \tag{22.4.4c}$$

In Eq. (22.4.4), *count* keeps track of the number of packets queued since the last drop. As implied by the equation, the probability P_a increases as *count* increases. This makes a drop increasingly likely as the time since the last drop increases. Once a packet is dropped, the *count* is reset to zero. With this approach, closely spaced packet drops are relatively less likely than widely spaced drops.

RED can be efficiently implemented in hardware with only a small number of add and shift instructions on each packet arrival. The implementation involves the efficient computation of average queue size, calculating the packet-dropping probability, and arriving at a decision on whether to drop a packet.

First, the average queue length can be calculated based on the following equation. Rearranging, Eq. (22.4.1), we get

$$Q_{\text{avg}} = Q_{\text{avg}} + w(Q_{\text{sample}} - Q_{\text{avg}}). \tag{22.4.5}$$

If w is chosen as a negative power of 2, i.e., $w = 2^{-n}$ where n is configurable. The advantage is that this can be implemented with a few shift operations and two additional instructions.

22.4.3 Setting $Q_{\min}$ and $Q_{\max}$

Consider the setting of values for $Q_{\min}$ and $Q_{\max}$. These values to a large extent are determined by average queue length, Q_{avg}. The choice of values for $Q_{\min}$ determines how efficiently the output link is utilized. If the traffic is fairly bursty, smaller values for $Q_{\min}$ will lead to packet dropping, thereby underutilizing the output link. As a result, $Q_{\min}$ should be chosen large enough such that the router will be able to absorb bursts as well as keep the link utilization at an acceptably high level.

The threshold $Q_{\max}$ determines the delay experienced by a packet as it transits through the router. A large value for $Q_{\max}$ means that more packets will be buffered in the queue ahead of the newly arrived packet, and they must be transmitted before the newly arrived packet. This introduces significant delay. Thus, the choice of values for $Q_{\max}$ depends on the maximum average delay that can be allowed by the router.

Also, the difference between the two thresholds $Q_{\max} - Q_{\min}$ should be larger than the typical increase in the calculated average queue length in one RTT. Given the traffix mix on today's Internet, a useful rule of thumb is to set $Q_{\max}$ to at least twice $Q_{\min}$. During periods of high load, since the average queue length is expected to vary between the two thresholds, there should be enough free buffer space above $Q_{\max}$ to absorb bursts in the traffic without forcing the router to enter tail drop mode.

22.5 Variations of RED

Due to the bursty nature of Internet traffic, the queues can fill up quickly and then become empty again. In such cases, the queue can be empty most of the time except during such period of burstiness. Furthermore, because the thresholds are compared with the average queue length, no dropping of packets takes place even when the instantaneous queue length is quite large.

22.5.1 Weighted Random Early Detection

The RED algorithm can be considered to be fair for all the flows passing through a single queue, where flows belong to the same traffic class. The flows from which the traffic needs to be discarded are chosen randomly without any bias. While fairness is desirable, often there are situations that need to introduce unfairness. To illustrate such needs, consider the following example of packet marking.

Example 22.3 *Packet marking.*

Consider the scenario where packets are marked as they enter the ISP network. The process of marking distinguishes packets as either out-of-profile or in-profile. The out-of-profile packets represent the excess traffic from a customer than the agreed bandwidth with the ISP. On the other hand, the in-profile packets confirm to the agreed bandwidth. The intention behind distinguishing the packets is to provide delivery guarantees for in-profile packets while out-of-profile packets are delivered best effort. When the ISP network is lightly loaded, out-of-profile packets would most likely get through the network, thereby using the spare bandwidth. However, during periods of congestion, these out-of-profile packets are dropped in preference to in-profile packets. ▲

The RED algorithm has been observed to be unfair in bandwidth sharing when there are different traffic classes [410]. Furthermore, as seen from the example, when congestion is anticipated, the router might need to enforce different packet-dropping behavior for different types of traffic (such as in-profile and out-of-profile packets). This translates into defining different drop profiles for different queues, or for different types of traffic in the same queue. Recall from Section 22.4 that a RED profile specifies the parameters $Q_{\min}$, $Q_{\max}$, $\overline{P}_{\max}$, and w that characterize the behavior of RED.

Weighted random early detection (WRED) attempts to achieve this by augmenting RED to use additional information from the packet. Such additional information in the packet can

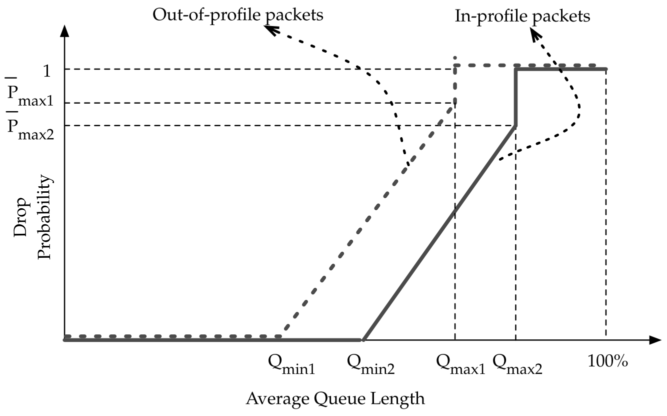


FIGURE 22.8 Weighted random early detection with in-profile and out-of-profile packets.

include but is not limited to source and destination address, source and destination ports, protocol flags and IP precedence/TOS bits. This information provides a context for the packet. Based on the context of the packet, a RED drop profile is chosen for packet dropping. For the example discussed earlier, an aggressive drop profile with higher dropping probability can be assigned for out-of-profile packets, while a less aggressive drop profile with lower dropping probability governs in-profile packets. Two such profiles are shown in Figure 22.8 where $Q_{\min 1}$, $Q_{\max 1}$ and $P_{\max 1}$ represent the aggressive drop profile, while the others with ($Q_{\max 2}$, $Q_{\max 2}$, and $\overline{P}_{\max 2}$) specify the less aggressive drop profile. In the figure, the aggressive profile is shown with dashed lines and the less aggressive profile with solid lines. Such modification of the drop probability depending on the context of the packet is referred as *weighting*; hence, this variation of RED is known as weighted RED or simply WRED.

On receiving packets, the router classifies them based on the contents of their headers using fast classification algorithms (discussed in Chapter 16) to establish context. This context affects how the packet is processed in the subsequent stages inside the router. When the packet arrives in the egress linecard, it is appended to the appropriate outgoing queue. Now the context information of the packet is used for choosing the appropriate drop profile, which determines how to drop the packet, if needed. Now let us look at two examples that illustrate how WRED is used in practice.

Example 22.4 *Assigning RED drop profiles to each queue.*

Consider a router that is configured to use three queues for sharing the output link O (Figure 22.9(a)). Each of these queues carries traffic of different priorities. The queue R carries high-priority traffic, while queues S and T carry medium-priority and low-priority traffic, respectively. Each queue shares a portion of the bandwidth of the outgoing link. For the high-priority traffic, the network administrator might be interested in dropping fewer packets compared to medium-priority traffic and lower-priority traffic. Thus, the administrator configures different drop profiles for each queue. In this case, a less aggressive drop profile is associated with queue R and a progressively more aggressive profile, one for each of the queues S and T.

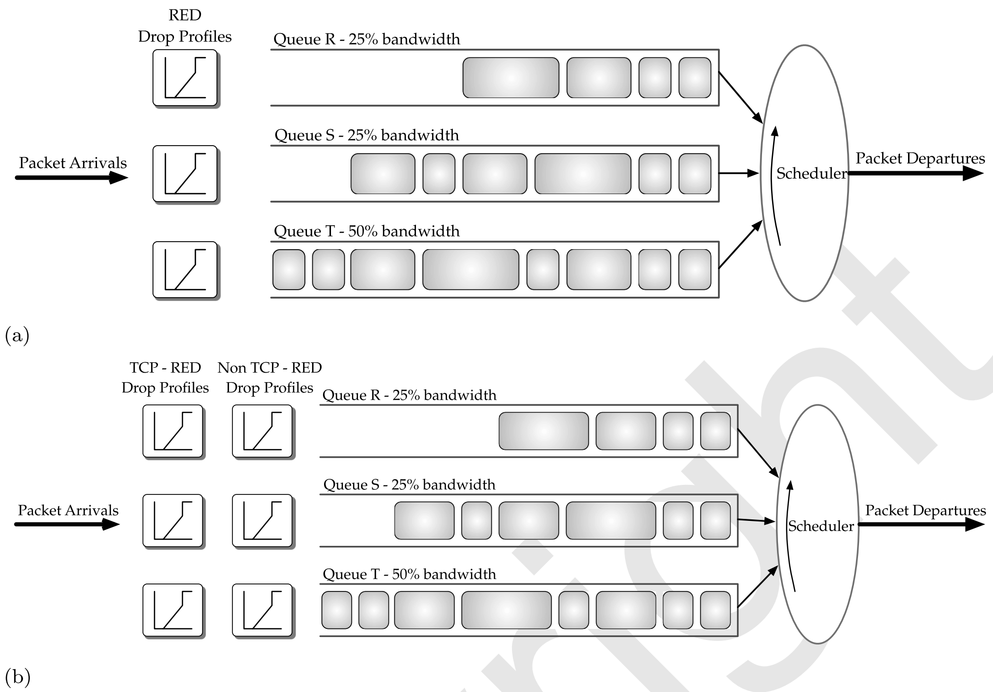


FIGURE 22.9 Two packet drop profiles for weighted random early detection (WRED).

Consider another scenario where the network administrator is interested in further distinguishing the traffic in each queue into TCP traffic and non-TCP traffic. Furthermore, he/she desires that in each queue more non-TCP traffic should be dropped rather than TCP traffic. In such cases, the administrator has to configure two drop profiles per queue with the appropriate drop probability. Depending on the type of the traffic (which is typically embedded in context of the packet), the appropriate drop profile will be applied. This scenario is illustrated in Figure 22.9(b). ▲

While WRED allows dropping of packets depending on their context, it does not dynamically adapt the dropping probability depending on amount of traffic. In the next section, we describe another variant of RED called adaptive RED that provides this flexibility.

22.5.2 Adaptive Random Early Detection

In basic RED, the decision about when to drop a packet is determined by the parameters in the RED profile—$Q_{\min}$, $Q_{\max}$, w, and $\overline{P}_{\max}$. However, choosing the appropriate values for these parameters has proven difficult as they are highly dependent on the nature and burstiness of the traffic. Let us look at a few scenarios on how the choice of these values affects the average latency experienced by a packet and the network throughput.

First, consider the scenario in which the output link is lightly congested and/or $\overline{P}_{\max}$ is configured to be high. In this case, Q_{avg} hovers close to $Q_{\min}$ and, thus, the queueing delay experienced by a packet will be low. Similarly, when the link is highly congested and/or $\overline{P}_{\max}$

is low, Q_{avg} operates closer to Q_{max} or even above Q_{max}. This results in a packet experiencing higher queueing delay.

Now consider another scenario in which an output link is congested due to a large number of connections flowing through it. In such cases, packets should be dropped from sufficiently large numbers of connections so that the offered traffic load from those sources is reduced considerably. However, overly aggressive drop behavior could result in underutilization of the network bandwidth as many more sources will back off their transmission rates. It has been observed that when there are N connections sharing the link, the effect of any given RED-induced packet drop is to reduce the offered load by a factor of $(1 - 1/2N)$. Thus, as N increases, the impact of a packet drop decreases and the RED algorithm needs to be aggressive. Otherwise, it would degenerate into simple tail drop. When N is small, the impact of packet drop is large and RED should be conservative. Otherwise, it will lead to underutilization of network bandwidth.

Example 22.5 *Impact of the number of connections on throughput*.

Assume that the output link capacity of 10 Mbps is shared equally among 100 simultaneous TCP connections. A RED-induced packet drop from a single connection reduces its transmission rate and therefore the offered load reduces to 9.95 Mbps. Alternatively, if only two connections share the link, RED-induced packet drop from one of the connection reduces the offered load to 7.5 Mbps. ▲

We can see from these scenarios that the average queueing delay and output link utilization are sensitive to the traffic load and, therefore, not predictable in advance. To achieve predictable average delays and maximum utilization of the outgoing link, the RED profile parameters must be constantly adjusted depending on the current traffic conditions.

Adaptive RED attempts to address these limitations by allowing RED to modify its parameters based on traffic load. The idea behind this algorithm is to adjust the value of $\overline{P}_{\text{max}}$ to restrict the average queue length Q_{avg} between Q_{min} and Q_{max}. In particular, it scales $\overline{P}_{\text{max}}$ by constant factors of α and β depending on whether it is less than Q_{min} or greater than Q_{max}. The algorithm is outlined below.

If the average queue length is less than Q_{min}, then RED is too aggressive in dropping packets. Thus, $\overline{P}_{\text{max}}$ is decreased by $\overline{P}_{\text{max}} \times \beta$. The decrease in drop probability reduces the number of packets being dropped, thus allowing the average queue length Q_{avg} to increase beyond Q_{min} but less than Q_{max}. In the other case, when the average queue length is greater than Q_{max}, then RED is too conservative and $\overline{P}_{\text{max}}$ is increased by $\overline{P}_{\text{max}} \times \alpha$. The increase in drop probability increases the number of dropped packets and reduces the average queue length Q_{avg} to less than Q_{max}. The relationship between the average queue length Q_{avg} and drop probability $\overline{P}_{\text{max}}$ is illustrated in Figure 22.10. The algorithm is run periodically using a predetermined time interval, and the constants α and β are supplied by the network operator.

To be more specific, recall that in basic RED, the choice for w that tracks the average queue length should be such that it filters transient congestion and still does not react too slowly to long-term congestion.

The rate at which this congestion could have built up depends partially on how many TCP connections pass through the outgoing link. If there are fewer connections sharing the link, then the congestion will build up relatively slowly and w should be low. However, a low

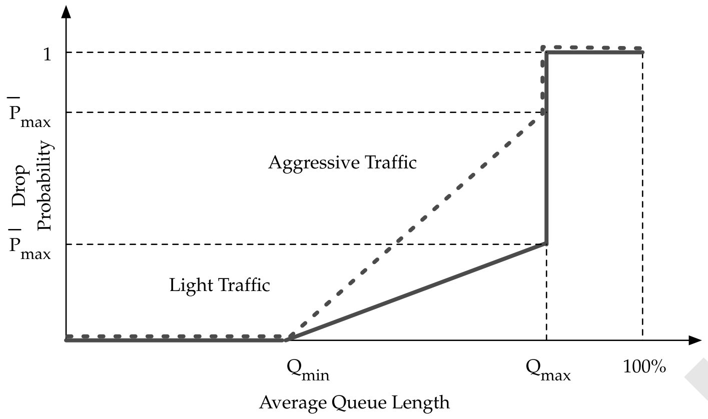


FIGURE 22.10 Adaptive random early detection (with different threshold values for $\overline{P}_{\max}$ for light and aggressive traffic).

value of *w* for a large number of connections would mean less aggressive dropping behavior, leading to queue overflow. Conversely, choosing a high value for *w* to enable fast RED behavior with many TCP connections can result in the drop behavior being overly aggressive in the presence of only a few flows. This leads to reduced output link utilization.

22.6 Explicit Feedback Schemes

Various schemes for packet dropping discussed in the previous section have been implicit, that is, use packet loss to indicate congestion. The packet losses are inferred at the end hosts and in response adjust their transmission rates. Such schemes do not require explicit help from the intermediate routers except the dropping of packets, which the router must do anyway when its queues are full. While these schemes are simple and easy to develop and deploy in a network, they suffer from a few disadvantages [737]:

- Dropping of packets wastes network resources used for carrying the packet from its source to the router experiencing congestion.
- Feedback schemes based on packet drops are based on an assumption about the network, interpreting packet loss as a sign of congestion. This might not be valid for all networks such as wireless networks.

A departure from these schemes is to explicitly communicate to the source when congestion occurs to reduce the transmission rate. Such explicit notifications can be categorized depending on the direction as:

- *Backward*—Here a congested router generates notification to the sender that travels in the opposite direction of the data traffic. Notification can be piggybacked on the header of the data packet bound for the sender (probably from the receiver), or a separate packet is generated and transmitted to the sender.

- *Forward*—In this case, the router generates notification to the sender, which flows in the direction of the traffic. This notification is carried by the data packets to the receiver. The receiver piggybacks the packets to the sender to echo the notification to the sender. Instead of echoing, the receiver can exercise the flow control at a higher layer protocol so that the sender reduces it transmission rate.

Now let us discuss these schemes in detail.

22.6.1 Choke Packets

In this approach, whenever a router experiences congestion, it sends explicit notifications to sources directly to reduce their transmission rate. These notifications could carry further details such as the percentage by which the source should reduce the rate or an upper bound on the rate the source should transmit. Routers implement a variation of this approach called *source quench*, which conveys to the source that the congestion has occurred and hence it should reduce the rate.

A source quench message could be generated by a router in two scenarios. In the first scenario, whenever a router discards a packet, it may send a source quench message. In the second scenario, the router may send the source quench message when the queue buffers exceed a certain threshold rather than waiting until the queues are full. This means that the packet that triggered the source quench message may be delivered to the destination. A source quench message is carried by ICMP.

The benefits of this approach are twofold. First, it provides the fastest feedback to the sources to reduce their rate. This has the benefit of reducing congestion as quickly as possible. Second, as the router that experiences congestion generates this feedback, the information is more precise. However, there are disadvantages from system design perspective. The router has to generate a new packet on the data path, which is typically implemented in hardware for high-speed requirements. To reduce such complexities in the hardware design, the data path informs the control processor to generate the source quench message, which is injected into the data path for forwarding it to the source.

22.6.2 Explicit Congestion Notification

Explicit congestion notification (ECN) is another feedback scheme that indicates congestion information by marking packets instead of dropping them. The destination of marked packets returns this congestion notification to the source. As a result, the source decreases its transmit rate. The implementation of ECN uses an ECN-specific field in the IP header with two bits—the ECN-capable Transport (ECT) bit and the Congestion Experienced (CE) bit. These two bits are mapped to bits 6 and 7 of the DSCP field in the IP header as shown in Figure 22.11. Both ECT and CE bits allow for four ECN field combinations—“00,” “01,” “10,” and “11.”

The ECN field combinations “01” and “10” are set by the data sender to indicate that the endpoints of the transport protocol are ECN capable. The non-ECT combination “00” indicates a packet that is not using ECN. The CE combination “11” is set by a router to indicate congestion to ECN-capable host systems. Table 22.1 summarizes each of the ECT and CE bit combination settings, including a brief note about how it is used.

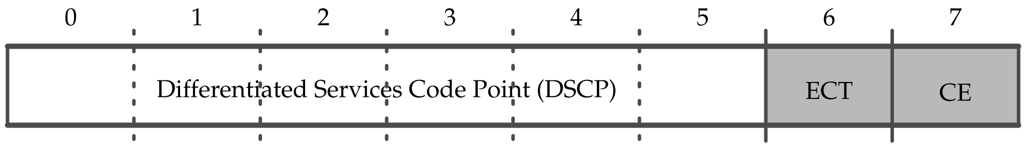


FIGURE 22.11 ECN field in the differentiated services field of IP packet header.

TABLE 22.1 ECN bit setting.

ECE Bit	CE bit	Remark
0	0	Not ECN-capable
0	1	ECN-capable end hosts
1	0	ENC-capable end hosts
1	1	Congestion experienced

The primary motivation for using two combinations for ECT is twofold. First, it allows a data sender to verify that intermediate routers and switches are not erasing the CE combination. Second, it allows a data source to check whether the data receivers are properly reporting to the sender the receipt of packets with the CE combination as required by the transport protocol.

To identify congestion and mark the packets, routers use the same algorithm as RED but modified to identify ECN field combinations. The modified algorithm is described as follows:

- If the queue occupancy is below Q_{min}, the packets are transmitted independently of whether ECN is enabled or not. This is similar to the normal RED algorithm.
- If the queue occupancy is between Q_{min} and Q_{max}, then the ECN field in the IP header is examined. Depending on the combination that occurs in the ECN field, one of the following scenarios can occur:
  - If the ECN field contains "00," then the end host systems are not ECN capable and the packet might be dropped based on the drop probability generated by the RED algorithm. This behavior is identical to normal RED behavior as described in Section 22.4.
  - If the ECN field contains either "01" or "10," then the RED algorithm determines whether the packet should have been dropped based on the drop probability. If the decision is to drop, then the ECN field is set to "11" and the packet is transmitted. Since ECN is enabled, the packet gets marked instead of being dropped.
  - If the ECN field contains "11," then the packet is transmitted. It means that some upstream router has experienced congestion and this information needs to be passed on further and ultimately indicated to the destination.
- If the queue occupancy is greater than Q_{max}, packets are dropped based on the drop probability generated by the RED algorithm. Again, this behavior is identical to the normal RED behavior.

For ECN to work, the end hosts using TCP and the intermediate routers should cooperate. When a packet arrives with the ECN field containing "11," the end host needs to recognize that congestion has occurred and invoke the congestion control algorithm, which must be the same as the congestion control response to a single *dropped* packet. Thus, when an ECN-capable source running TCP receives a packet with the CE combination set, it should respond by halving its congestion window. Such behavior is required for the incremental deployment of ECN in end systems and routers.

ECN is similar to an earlier scheme known as the DECbit scheme [575]; however, there are quite a few differences. In the DECbit scheme, the algorithm for generating the notification is coupled with the actual mechanics of how to carry notification. However, ECN just outlines how the congestion notification is carried to the destination and back to the source in IP networks. It is not tied to any queue management algorithm and is typically used in conjunction with RED. Further, the DECbit scheme uses only a single bit to indicate congestion as opposed to two bits in ECN.

While the key benefits of ECN are similar to RED, there are additional benefits that ECN provides. Unlike RED, which drops packets and wastes network resources used to forward packets to the point where they experience congestion, ECN, by marking the packets, avoids such resource waste. With ECN, TCP hosts can distinguish between packet loss due to transmission errors and congestion signals. This could be especially useful in networks with relatively high bit error rates. Also, since ECN uses bits in the packet header carrying data, it does not add any additional traffic to an already congested network. Furthermore, ECN provides flexibility to deploy incrementally in the Internet.

22.7 New Class of Algorithms

Even though RED is an improvement over traditional drop tail policies, there are a few shortcomings. The fundamental problem in RED is the use of queue length as an indicator of congestion in the network and the assumption that larger queue length means larger number of flows are competing for the bandwidth of the outgoing link. This need not be true. For instance, a single flow can transmit at a rate greater than the bottleneck output link capacity during a brief busy period. This could result in the buildup of queues that are easily as large as the queue length with large number of flows. If the packet interarrival time follows the exponential distribution (see Appendix B.10) the queue length relates directly to the number of flows—a well-known result in queueing theory. However, this is not the case with Internet and, thus, RED requires tuning a range of parameters to operate correctly under different congestion scenarios.

Thus, instead of using the average queue length, the stochastic fair blue (SFB) algorithm [222] is based on the history of the packet loss and link utilization for managing congestion. Unlike RED, SFB maintains only a single probability P—this is used for deciding whether an incoming packet should be dropped. If the queue continues to drop packets due to buffer overflow, the drop probability is increased; it is decreased as the queue becomes empty or the output link is idle.

The active queue management schemes discussed so far rely on end hosts being cooperative. In other words, the end hosts react to congestion feedback by slowing the transmission rate similar to TCP. However, new classes of applications such as video streaming do not always use TCP-like congestion control and, thus, do not react to the congestion feedback from the network. Such applications could essentially consume an unfair amount of bandwidth affecting responsive flows and leading to a high packet loss rate in the network. This effectively reduces the throughput of the network. The idea behind these approaches is to detect *unresponsive flows* and to limit their rates so that they do not impact the performance of responsive flows [226].

You may note that datagram congestion control protocol (DCCP) [371] has recently been proposed in the standards track in IETF. DCCP is meant for use in UDP-based applications that might require timeliness as well as reliable delivery; for that, TCP-friendly rate control and TCP-like congestion control have been proposed—see [229], [230].

22.8 Analyzing System Behavior

From the discussions presented so far, you can see that a combination of features might be employed at a router that spans from scheduling to congestion control. This raises an important question: how do we know one scheme is better than another, and more importantly, when multiple features are in place, how do we know whether the overall system is behaving as intended?

From our earlier discussion, we know that a major trade-off to consider with any scheduling discipline is to understand if it is fair to different traffic classes, yet can be implemented with low time complexity. Typically, in this trade-off, low time complexity is preferable from a router implementation point of view while the solution might not be fully optimal from the scheduling discipline point of view. Thus, understanding the details of such a trade-off, the time complexity analysis is important. Furthermore, it is important to get an idea on performance impact. Since packet arrival is of a bursty nature, developing analytical models is typically difficult. Thus, often such analysis entails doing a simulation model-based analysis where realistic traffic streams are mimicked, and then doing a systematic assessment of the behavior observed based on output of such simulations.

Similarly, to understand any implicit or explicit feedback mechanisms, performance analysis is required. Typically in such analysis, traffic sources that follow TCP congestion control mechanisms are considered. Since the overall analysis requires knowing how a router handles congestions, often a network topology is needed for such studies. A commonly used topology for such studies is known as the *dumbbell topology*—this means that several TCP sessions share a bandwidth-limited bottleneck link with two routers [226]. You may note that most such analysis avoids bringing the routing component into the picture. Why so? First, in such studies, the goal is to understand the basic behavior without complicating it by considering too many dimensions. Second, in many instances, it is fair to assume that routing would not change in a short window time frame during which one might be interested in congestion behavior or buffer management. Yet, it is important to understand the relation between congestion control and routing, for example, if there is a link or a router failure—such analysis using simulation or a simulated environment, however, requires consideration of many parameters or factors in a systematic manner [64], [581], [635].

What tools can you use for such analysis? From a simulation point of view, ns-2 [525] is a widely popular public domain platform. There are commercial tools available from vendors such as OPNET [536] that allow simulation of control mechanisms along with routing. In either type of platform, to try out a new area, writing a new software code segment is often necessary. Certainly, any ability to test a concept through a virtual platform that operates on the Internet is important. Several such approaches are currently ongoing [535], [568], [722].

22.9 Summary

We started this chapter with a discussion of different queueing disciplines that a router might employ for scheduling packets. Critical issues are whether a discipline is fair to different traffic streams, yet packet processing can be done efficiently. In this regard, deficit round-robin and modified deficit round-robin schemes are found to be most preferred. It is, however, important to note that most scheduling algorithms base their decisions using flow-level information in order to provide flow-based "fairness." On the other hand, there are several newer classes of applications that create multiple TCP flows in parallel. Thus, if we were to consider at the level of application session-level fairness, application sessions with parallel TCP flows can receive unduely higher fairness. For such situations, how a router can provide application-level fairness is still an on-going research problem.

Congestion in routers can occur due to various factors. Thus, schemes are needed to handle congestion, which can be broadly divided into implicit feedback schemes and explicit feedback schemes. The implicit feedback schemes rely on the local observations at the end-hosts to infer about congestion in the network, and drop packets at routers. However, explicit feedback schemes rely on conveying the congestion notification explicitly.

We ended the chapter by including a brief discussion on how to analyze system behavior when there are a variety of factors to consider in regard to congestion control and routers.

Finally, it is important to realize that there is a direct connection between how well a network is traffic engineered along with buffer sizing and the congestion perceived by users. For example, from the viewpoint of an ISP, its goal is to serve its customers with good quality of service, which may be based on service level agreements; thus, the ISP wants to engineer its network at an adequate performance level. For instance, many providers use 95th percentile of traffic volume to engineer their networks. Thus, unless there is a sudden surge in traffic due to unanticipated demand, most well-engineered networks should be able to avoid network level congestions. This is important to keep in mind.

Further Lookup

The literature on packet queueing and scheduling and active queue management is vast. We highlight only a few examples here. For a general processing sharing principle, refer to [543], [544]. For various queueing disciplines, see [72], [378], [379], [640], [670].

For early works on congestion control in packet networks, see [52], [575], [508]. For active queue management, through mechanisms such as RED and its many variations, see [88], [222], [227], [295], [343], [495], [410], [762]. It is worth noting that if a buffer is small, RED might not lead to improvement in network performance [451]; thus, it is important to size the buffer properly for traffic engineering (refer to Section 7.2.3). For understanding the interaction of congestion control and routing, refer to [64], [288], [581], [598], [635].

For the recently proposed datagram congestion control protocol (DCCP) and its TCP-friendly rate control and TCP-like congestion control mechanisms, see [229], [230], [371].

Finally, readers might be interested in comparing congestion control mechanisms used for voice call traffic in the PSTN, discussed in Section 11.6, with the ones discussed in this chapter for the Internet.

Exercises

22.1 Explain different packet scheduling disciplines, their strengths and limitations, and deficit round-robin scheme.

22.2 Consider Example 22.2; work through the numbers if the quantum credit for each queue is reduced to 300.

22.3 Explain why it is important to consider TCP congestion control mechanisms in a router design.

22.4 Explain strengths and limitations of the random early detection mechanism and its variations.