White Paper

Supporting Differentiated Service Classes: Queue Scheduling Disciplines

Chuck Semeria
Marketing Engineer

Juniper Networks, Inc.
1194 North Mathilda Avenue
Sunnyvale, CA 94089 USA
408 745 2000 or 888 JUNIPER
www.juniper.net

Part Number:: 200020-001 12/01

Contents

List of Figures

Executive Summary

This is part of a series of publications from Juniper Networks that describe the mechanisms that allow you to support differentiated service classes in large IP networks. This paper provides an overview of a number of popular queue scheduling disciplines: first-in-first-out queuing (FIFO), priority queuing (PQ), fair queuing (FQ), weighted fair queuing (WFQ), weighted round-robin queuing (WRR)—or class-based queuing (CBQ) as it is sometimes called—and deficit weighted round robin queuing (DWRR). The other papers in this series provide technical discussions of active queue memory management, host TCP congestion-avoidance mechanisms, and a number of other issues related to the deployment of differentiated service classes in your network.

Perspective

A router is a shared resource in your network. Many of the problems that you face in your network are related to the allocation of a limited amount of shared resources (buffer memory and output port bandwidth) to competing users, applications, and service classes. A queue scheduling discipline allows you to manage access to a fixed amount of output port bandwidth by selecting the next packet that is transmitted on a port. As you will see, there are many different queue scheduling disciplines, each attempting to find the correct balance between complexity, control, and fairness. The queue scheduling disciplines supported by router vendors are implementation-specific, which means that there are no real industry standards. However, a vendor sometimes refers to its own implementation as *the* standard. The difficulty with recognizing any implementation as a standard is that each vendor combines features from a number of well-known generic queue scheduling disciplines to provide the total functionality that they believe their customers need to manage output port congestion.

Congestion occurs when packets arrive at an output port faster than they can be transmitted. This means that a router interface is considered mildly congested if just a single packet has to wait for another packet to complete its transmission. The amount of delay introduced by congestion can range anywhere from the amount of time that it takes to transmit the last bit of the previously arriving packet to infinity, where the packet is dropped due to buffer exhaustion. It is important to minimize the amount of congestion in your network, because congestion reduces packet throughput, increases end-to-end delay, causes jitter, and can lead to packet loss if there is insufficient buffer memory to store all of the packets that are waiting to be transmitted.

There are several tasks that any queue scheduling discipline should accomplish:

- Support the fair distribution of bandwidth to each of the different service classes competing for bandwidth on the output port. If certain service classes are required to receive a larger share of bandwidth than other service classes, fairness can be supported by assigning weights to each of the different service classes.
- Furnish protection (firewalls) between the different service classes on an output port, so that a poorly behaved service class in one queue cannot impact the bandwidth and delay delivered to other service classes assigned to other queues on the same output port.
- Allow other service classes to access bandwidth that is assigned to a given service class if the given service class is not using all of its allocated bandwidth.
- Provide an algorithm that can be implemented in hardware, so it can arbitrate access to bandwidth on the highest-speed router interfaces without negatively impacting system forwarding performance. If the queue scheduling discipline cannot be implemented in hardware, then it can be used only on the lowest-speed router interfaces, where the reduced traffic volume does not place undue stress on the software implementation.

When considering the deployment of differentiated service classes in your network, you need to understand the capabilities that each router vendor supports and why they made their specific design decisions. This paper describes the operation, benefits, limitations, and applications for a number of classic queue scheduling disciplines to help you better understand the range of options that are available. The traditional queue scheduling disciplines that we'll consider here include:

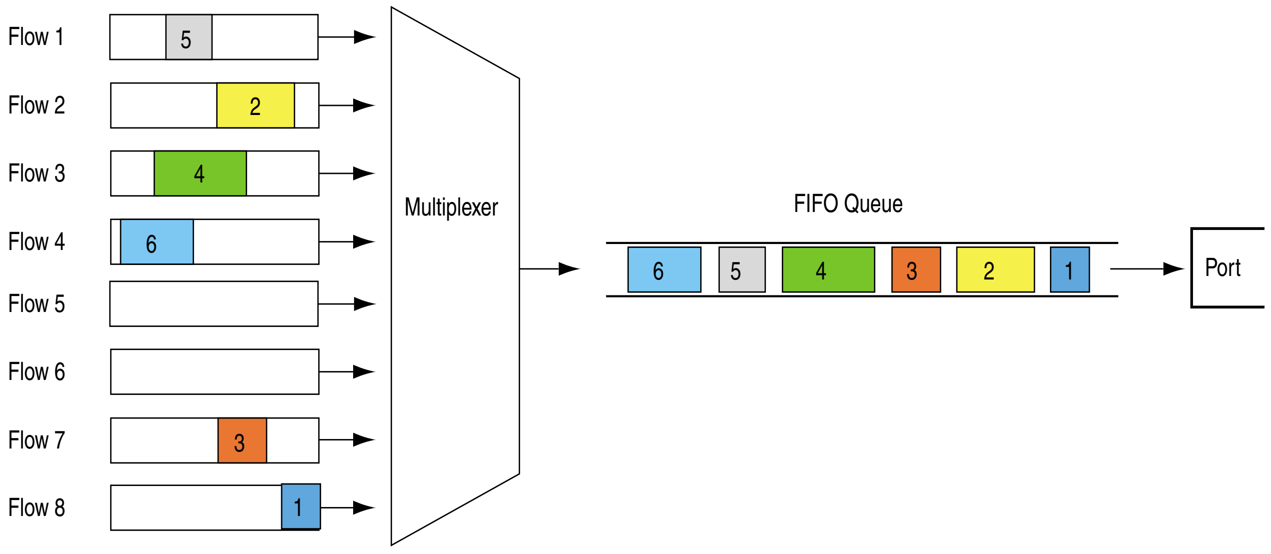
- First-in-first-out queuing (FIFO),
- Priority queuing (PQ),
- Fair queuing (FQ),
- Weighted fair queuing (WFQ),
- Weighted round-robin queuing (WRR), also known as class-based queuing (CBQ), and
- Deficit weighted round robin queuing (DWRR)

Router vendors use various combinations of these terms to describe their implementations. Be careful not to confuse the theoretical scheduling algorithms described in this paper with a specific vendor's implementation, and do not make the assumption that all queue scheduling disciplines behave as their names might suggest. Rather than simply relying on the name that the vendor uses to describe its queue scheduling discipline, you need to carefully examine each vendor's implementation, so that you can fully understand how it operates and determine if it provides the features that you need to support your subscriber requirements.

First-in, First-out (FIFO) Queuing

First-in, first-out (FIFO) queuing is the most basic queue scheduling discipline. In FIFO queuing, all packets are treated equally by placing them into a single queue, and then servicing them in the same order that they were placed into the queue. FIFO queuing is also referred to as First-come, first-served (FCFS) queuing. (See Figure 1.)

Figure 1: First-in, First-out (FIFO) Queuing



FIFO Benefits and Limitations

FIFO queuing offers the following benefits:

- For software-based routers, FIFO queuing places an extremely low computational load on the system when compared with more elaborate queue scheduling disciplines.
- The behavior of a FIFO queue is very predictable—packets are not reordered and the maximum delay is determined by the maximum depth of the queue.
- As long as the queue depth remains short, FIFO queuing provides simple contention resolution for network resources without adding significantly to the queuing delay experienced at each hop.

FIFO queuing also poses the following limitations:

- A single FIFO queue does not allow routers to organize buffered packets, and then service one class of traffic differently from other classes of traffic.
- A single FIFO queue impacts all flows equally, because the mean queuing delay for all flows increases as congestion increases. As a result, FIFO queuing can result in increased delay, jitter, and loss for real-time applications traversing a FIFO queue.
- During periods of congestion, FIFO queuing benefits UDP flows over TCP flows. When experiencing packet loss due to congestion, TCP-based applications reduce their transmission rate, but UDP-based applications remain oblivious to packet loss and continue transmitting packets at their usual rate. Because TCP-based applications slow their transmission rate to adapt to changing network conditions, FIFO queuing can result in increased delay, jitter, and a reduction in the amount of output bandwidth consumed by TCP applications traversing the queue.
- A bursty flow can consume the entire buffer space of a FIFO queue, and that causes all other flows to be denied service until after the burst is serviced. This can result in increased delay, jitter, and loss for the other well-behaved TCP and UDP flows traversing the queue.

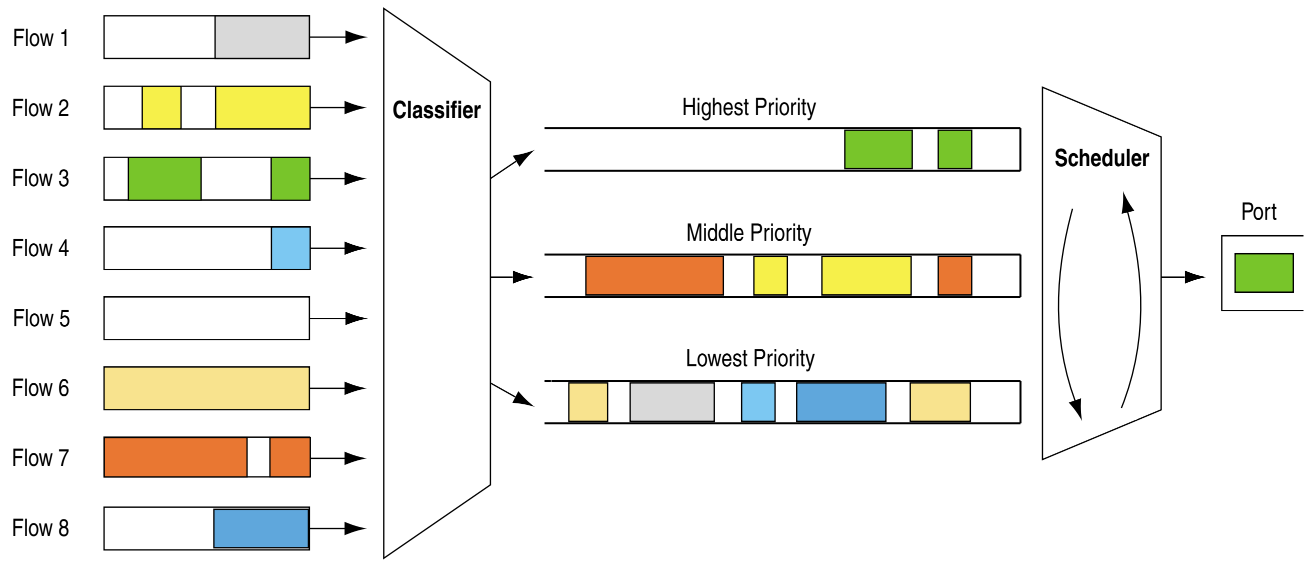
FIFO Implementations and Applications

Generally, FIFO queuing is supported on an output port when no other queue scheduling discipline is configured. In some cases, router vendors implement two queues on an output port when no other queue scheduling discipline is configured: a high-priority queue that is dedicated to scheduling network control traffic and a FIFO queue that schedules all other types of traffic.

Priority Queuing (PQ)

Priority queuing (PQ) is the basis for a class of queue scheduling algorithms that are designed to provide a relatively simple method of supporting differentiated service classes. In classic PQ, packets are first classified by the system and then placed into different priority queues. Packets are scheduled from the head of a given queue only if all queues of higher priority are empty. Within each of the priority queues, packets are scheduled in FIFO order. (See Figure 2.)

Figure 2: Priority Queuing



PQ Benefits and Limitations

PQ offers a couple of benefits:

- For software-based routers, PQ places a relatively low computational load on the system when compared with more elaborate queuing disciplines.
- PQ allows routers to organize buffered packets, and then service one class of traffic differently from other classes of traffic. For example, you can set priorities so that real-time applications, such as interactive voice and video, get priority over applications that do not operate in real time.

But PQ also results in several limitations:

- If the amount of high-priority traffic is not policed or conditioned at the edges of the network, lower-priority traffic may experience excessive delay as it waits for unbounded higher-priority traffic to be serviced.
- If the volume of higher-priority traffic becomes excessive, lower-priority traffic can be dropped as the buffer space allocated to low-priority queues starts to overflow. If this occurs, it is possible that the combination of packet dropping, increased latency, and packet retransmission by host systems can ultimately lead to complete resource starvation for lower-priority traffic. Strict PQ can create a network environment where a reduction in the quality of service delivered to the highest-priority service is delayed until the entire network is devoted to processing only the highest-priority service class.
- A misbehaving high-priority flow can add significantly to the amount of delay and jitter experienced by other high-priority flows sharing the same queue.
- PQ is not a solution to overcome the limitation of FIFO queuing where UDP flows are favored over TCP flows during periods of congestion. If you attempt to use PQ to place TCP flows into a higher-priority queue than UDP flows, TCP window management and flow control mechanisms will attempt to consume all of the available bandwidth on the output port, thus starving your lower-priority UDP flows.

PQ Implementations and Applications

Typically, router vendors allow PQ to be configured to operate in one of two modes:

- Strict priority queuing
- Rate-controlled priority queuing

Strict PQ ensures that packets in a high-priority queue are always scheduled before packets in lower-priority queues. Of course, the challenge with this approach is that an excessive amount of high-priority traffic can cause bandwidth starvation for lower priority service classes. However, some carriers may actually want their networks to support this type of behavior. For example, assume a regulatory agency requires that, in order to carry VoIP traffic, a service provider must agree (under penalty of a heavy fine) not to drop VoIP traffic in order to guarantee a uniform quality of service, no matter how much congestion the network might experience. The congestion could result from imprecise admission control leading to an excessive amount of VoIP traffic or, possibly, a network failure. This behavior can be supported by using strict PQ without a bandwidth limitation, placing VoIP traffic in the highest-priority queue, and allowing the VoIP queue to consume bandwidth that would normally be allocated to the lower-priority queues, if necessary. A provider might be willing to support this type of behavior if the penalties imposed by the regulatory agency exceed the rebates it is required to provide other subscribers for diminished service.

Rate-controlled PQ allows packets in a high-priority queue to be scheduled before packets in lower-priority queues only if the amount of traffic in the high-priority queue stays below a user-configured threshold. For example, assume that a high-priority queue has been rate-limited to 20 percent of the output port bandwidth. As long as the high-priority queue consumes less than 20 percent of the output port bandwidth, packets from this queue are scheduled ahead of packets from lower-priority queues. However, if the high-priority queue consumes more than 20 percent of the output port bandwidth, packets from lower-priority queues can be scheduled ahead of packets from the high-priority queue. When this occurs, there are no standards, so each vendor determines how its implementation schedules lower-priority packets ahead of high-priority packets.

There are two primary applications for PQ at the edges and in the core of your network:

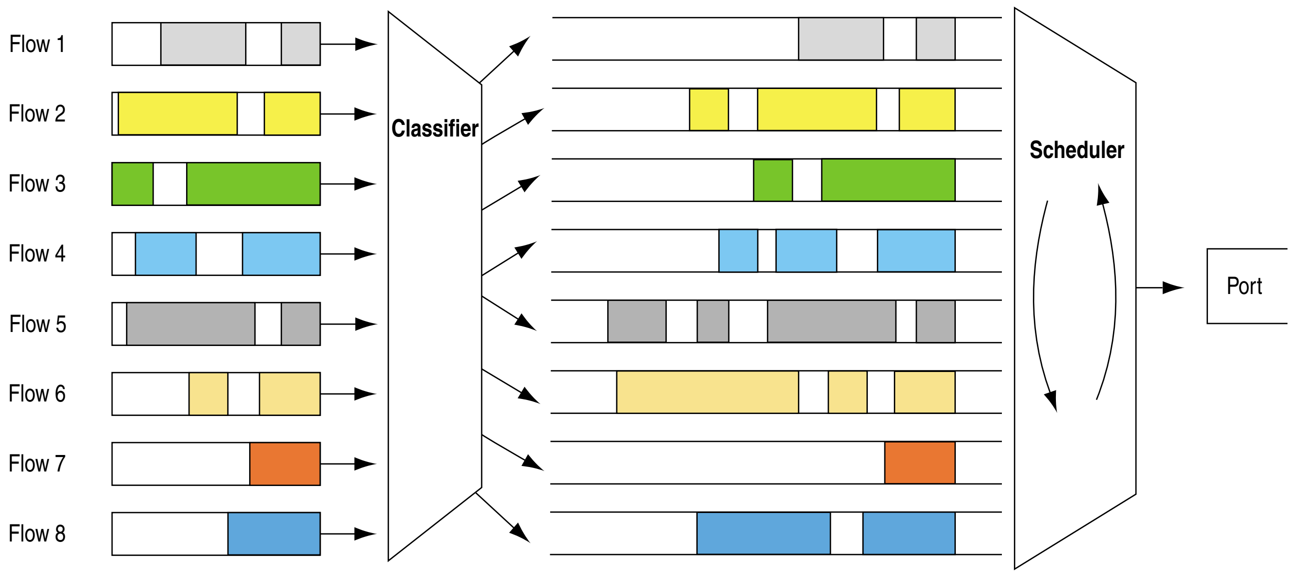
- PQ can enhance network stability during periods of congestion by allowing you to assign routing-protocol and other types of network-control traffic to the highest-priority queue.
- PQ supports the delivery of a high-throughput, low-delay, low-jitter, and low-loss service class. This capability allows you to deliver real-time applications, such as interactive voice or video, or to support TDM circuit emulation or SNA traffic by giving priority to these services before all other applications.

However, support for these types of services requires that you effectively condition traffic at the edges of your network to prevent high-priority queues from becoming oversubscribed. If you neglect this part of the design process, you will discover that it is impossible to support these services. The real challenge lies in the fact that it is much easier to condition traffic and allocate bandwidth to a queue for certain applications than for other applications. For example, it is much easier to provision resources for a well-defined application, such as VoIP, where you know the packet size, traffic volume, and traffic behavior, than it is to provision resources for other types of applications, such as interactive video, where there are just too many variables. It is the presence of these unknowns that makes it extremely difficult to configure traffic conditioning thresholds, maximum queue depths, and bandwidth limits for high-priority queues.

Fair Queuing (FQ)

Fair queuing (FQ) was proposed by John Nagle in 1987. FQ is the foundation for a class of queue scheduling disciplines that are designed to ensure that each flow has fair access to network resources and to prevent a bursty flow from consuming more than its fair share of output port bandwidth. In FQ, packets are first classified into flows by the system and then assigned to a queue that is specifically dedicated to that flow. Queues are then serviced one packet at a time in round-robin order. Empty queues are skipped. FQ is also referred to as per-flow or flow-based queuing. (See Figure 3.)

Figure 3: Fair Queuing (FQ)



FQ Benefits and Limitations

The primary benefit of FQ is that an extremely bursty or misbehaving flow does not degrade the quality of service delivered to other flows, because each flow is isolated into its own queue. If a flow attempts to consume more than its fair share of bandwidth, then only its queue is affected, so there is no impact on the performance of the other queues on the shared output port.

FQ also involves several limitations:

- Vendor implementations of FQ are implemented in software, not hardware. This limits the application of FQ to low-speed interfaces at the edges of the network.
- The objective of FQ is to allocate the same amount of bandwidth to each flow over time. FQ is not designed to support a number of flows with different bandwidth requirements.
- FQ provides equal amounts of bandwidth to each flow only if all of the packets in all of the queues are the same size. Flows containing mostly large packets get a larger share of output port bandwidth than flows containing predominantly small packets.
- FQ is sensitive to the order of packet arrivals. If a packet arrives in an empty queue immediately after the queue is visited by the round-robin scheduler, the packet has to wait in the queue until all of the other queues have been serviced before it can be transmitted.

- FQ does not provide a mechanism that allows you to easily support real-time services, such as VoIP.
- FQ assumes that you can easily classify network traffic into well-defined flows. In an IP network, this is not as easy as it might first appear. You can classify flows based on a packet's source address, but then each workstation is given the same amount of network resources as a server or mainframe. If you attempt to classify flows based on the TCP connection, then you have to look deeper into the packet header, and you still have to deal with other issues resulting from encryption, fragmentation, and UDP flows. Finally, you might consider classifying flows based on source/destination address pairs. This gives an advantage to servers that have many different sessions, but still provides more than a fair share of network resources to multitasking workstations.
- Depending on the specific mechanism you use to classify packets into flows, FQ generally cannot be configured on core routers, because a core router would be required to support thousands or tens of thousands of discrete queues on each port. This increases complexity and management overhead, which adversely impacts the scalability of FQ in large IP networks.

FQ Implementations and Applications

FQ is typically applied at the edges of the network, where subscribers connect to their service provider. Vendor implementations of FQ typically classify packets into 256, 512, or 1024 queues using a hash function that is calculated across the source/destination address pair, the source/destination UDP/TCP port numbers, and the IP ToS byte. FQ requires minimal configuration (it is either enabled or disabled) and is self-optimizing—each of the n active queues is allocated $1/n$ of the output port bandwidth. As the number of queues changes, the bandwidth allocated to each of the queues changes. For example, if the number of queues increases from n to $(n+1)$, then the amount of bandwidth allocated to each of the queues is decreased from $1/n$ of the output port bandwidth to $1/(n+1)$ of the output port bandwidth.

FQ provides excellent isolation for individual traffic flows because, at the edges of the network, a typical subscriber has a limited number of flows, so each flow can be assigned to a dedicated queue, or else a very small number of flows, at most, are assigned to each queue. This reduces the impact that a single misbehaving flow can have on all of the other flows traversing the same output port.

In class-based FQ, the output port is divided into a number of different service classes. Each service class is allocated a user-configured percentage of the output port bandwidth. Then, within the bandwidth block allocated to each of the service classes, FQ is applied. As a result, all of the flows assigned to a given service class are provided equal shares of the aggregate bandwidth configured for that specific service class. (See Figure 4.)

Figure 4: Class-based Fair Queuing

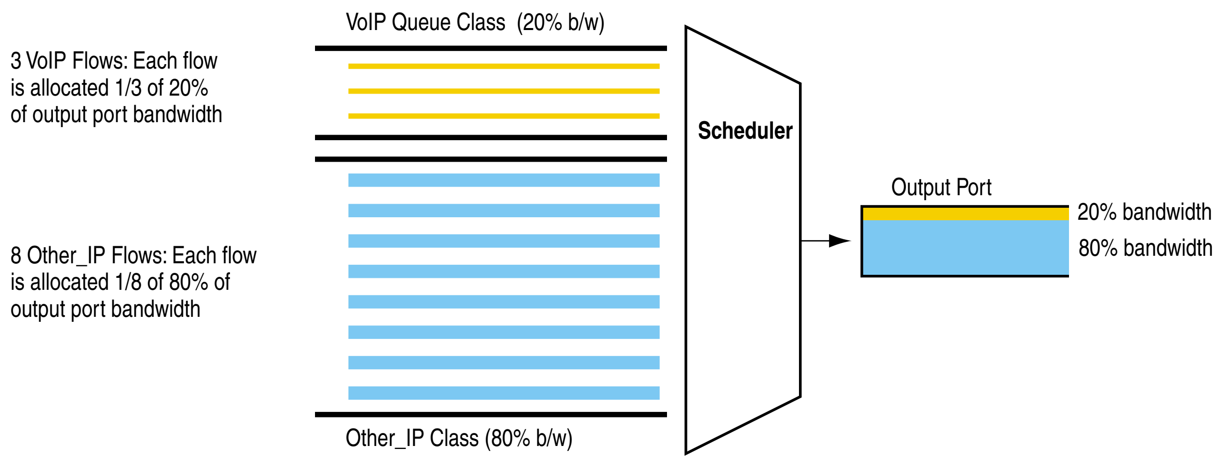

VoIP Queue Class (20% b/w)
3 VoIP Flows: Each flow is allocated 1/3 of 20% of output port bandwidth
8 Other_IP Flows: Each flow is allocated 1/8 of 80% of output port bandwidth
Other_IP Class (80% b/w)
Scheduler
Output Port
20% bandwidth
80% bandwidth

Figure 4 assumes that two service classes are configured for an output port—VoIP is allocated 20 percent of the output port bandwidth and Other_IP is allocated 80 percent. In the class-based FQ model, each VoIP flow is allocated an equal amount of the 20 percent bandwidth block, each Other_IP flow is allocated an equal amount of the 80 percent block, thus VoIP flows do not interfere with Other_IP flows nor visa versa.

Weighted Fair Queuing (WFQ)

Weighted fair queuing (WFQ) was developed independently in 1989 by Lixia Zhang and by Alan Demers, Srinivasan Keshav, and Scott Shenke. WFQ is the basis for a class of queue scheduling disciplines that are designed to address limitations of the FQ model:

- WFQ supports flows with different bandwidth requirements by giving each queue a weight that assigns it a different percentage of output port bandwidth.
- WFQ also supports variable-length packets, so that flows with larger packets are not allocated more bandwidth than flows with smaller packets. Supporting the fair allocation of bandwidth when forwarding variable-length packets adds significantly to the computational complexity of the queue scheduling algorithm. This is the primary reason that queue scheduling disciplines have been much easier to implement in fixed-length, cell-based ATM networks than in variable-length, packet-based IP networks.

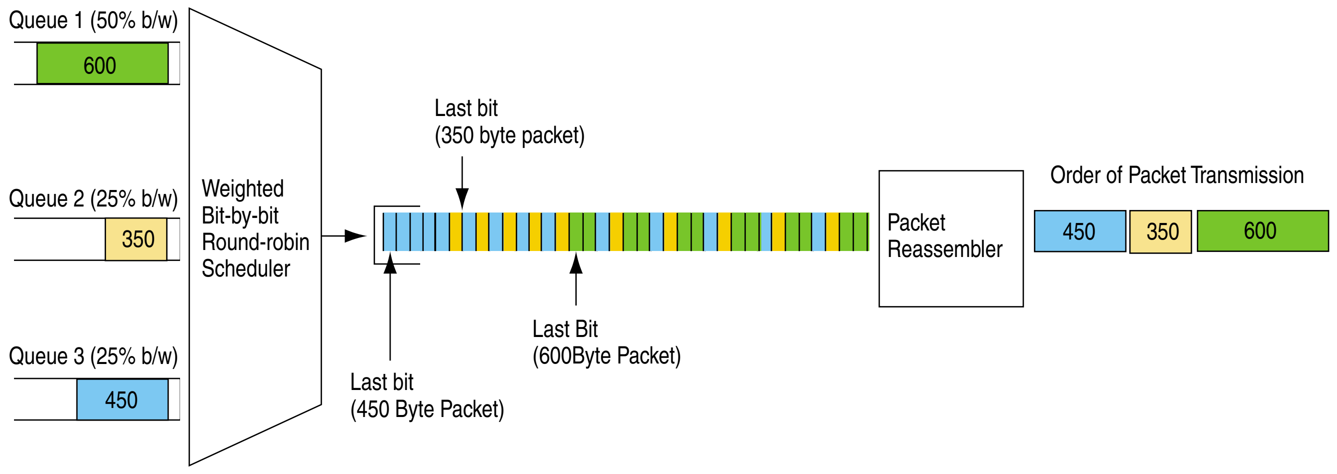
In 1992, A.K.J. Parekh proved that for sessions that are shaped at the edges of the network by token or leaky bucket rate control, WFQ can provide strong upper-bound, end-to-end delay performance guarantees.

WFQ Algorithm

WFQ supports the fair distribution of bandwidth for variable-length packets by approximating a generalized processor sharing (GPS) system. While GPS is a theoretical scheduler that cannot be implemented, its behavior is similar to a weighted bit-by-bit round-robin scheduling discipline. In a weighted bit-by-bit round-robin scheduling discipline the individual bits from packets at the head of each queue are transmitted in a WRR manner. This approach supports the fair allocation of bandwidth, because it takes packet length into account. As a result, at any moment in time, each queue receives its configured share of output port bandwidth. Although transmitting packets from different queues one bit at a time can be supported by a TDM network, it cannot be supported by a statistically multiplexed network. However, if you imagine the placement of a packet reassembler at the far end of the link, the order in which each packet would eventually be fully assembled is determined by the order in which the last bit of each packet is transmitted. This is referred to as the packet's finish time.

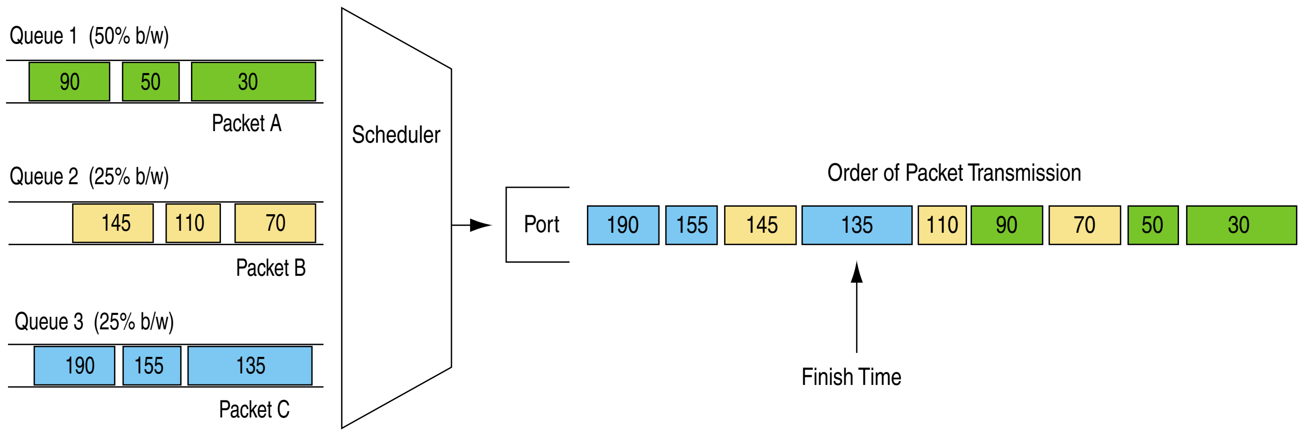
Figure 5 shows a weighted bit-by-bit round-robin scheduler servicing three queues. Assume that queue 1 is assigned 50 percent of the output port bandwidth and that queue 2 and queue 3 are each assigned 25 percent of the bandwidth. The scheduler transmits two bits from queue 1, one bit from queue 2, one bit from queue 3, and then returns to queue 1. As a result of the weighted scheduling discipline, the last bit of the 600-byte packet is transmitted before the last bit of the 350-byte packet, and the last bit of the 350-byte packet is transmitted before the last bit of the 450-byte packet. This causes the 600-byte packet to finish (complete reassembly) before the 350-byte packet, and the 350-byte packet to finish before the 450-byte packet.

Figure 5: A Weighted Bit-by-bit Round-robin Scheduler with a Packet Reassembler


Queue 1 (50% b/w)
600
Queue 2 (25% b/w)
350
Queue 3 (25% b/w)
450
Weighted Bit-by-bit Round-robin Scheduler
Last bit (350 byte packet)
Last Bit (600Byte Packet)
Last bit (450 Byte Packet)
Packet Reassembler
Order of Packet Transmission
450
350
600

WFQ approximates this theoretical scheduling discipline by calculating and assigning a finish time to each packet. Given the bit rate of the output port, the number of active queues, the relative weight assigned to each of the queues, and the length of each of the packets in each of the queues, it is possible for the scheduling discipline to calculate and assign a finish time to each arriving packet. The scheduler then selects and forwards the packet that has the earliest (smallest) finish time from among all of the queued packets. It is important to understand that the finish time is not the actual transmission time for each packet. Instead, the finish time is a number assigned to each packet that represents the order in which packets should be transmitted on the output port (Figure 6).

Figure 6: Weighted Fair Queuing (WFQ)—Service According to Packet Finish Time



When each packet is classified and placed into its queue, the scheduler calculates and assigns a finish time for the packet. As the WFQ scheduler services its queues, it selects the packet with the earliest (smallest) finish time as the next packet for transmission on the output port. For example, if WFQ determines that packet A has a finish time of 30, packet B has a finish time of 70, and packet C has a finish time of 135, then packet A is transmitted before packet B or packet C. In Figure 6, observe that the appropriate weighting of queues allows a WFQ scheduler to transmit two or more consecutive packets from the same queue.

WFQ Benefits and Limitations

Weighted fair queuing has two primary benefits:

- WFQ provides protection to each service class by ensuring a minimum level of output port bandwidth independent of the behavior of other service classes.
- When combined with traffic conditioning at the edges of a network, WFQ guarantees a weighted fair share of output port bandwidth to each service class with a bounded delay.

However, weighted fair queuing comes with several limitations:

- Vendor implementations of WFQ are implemented in software, not hardware. This limits the application of WFQ to low-speed interfaces at the edges of the network.
- Highly aggregated service classes means that a misbehaving flow within the service class can impact the performance of other flows within the same service class.
- WFQ implements a complex algorithm that requires the maintenance of a significant amount of per-service class state and iterative scans of state on each packet arrival and departure.
- Computational complexity impacts the scalability of WFQ when attempting to support a large number of service classes on high-speed interfaces.
- On high-speed interfaces, minimizing delay to the granularity of a single packet transmission may not be worth the computational expense if you consider the insignificant amount of serialization delay introduced by high-speed links and the lower computational requirements of other queue scheduling disciplines.
- Finally, even though the guaranteed delay bounds supported by WFQ may be better than for other queue scheduling disciplines, the delay bounds can still be quite large.

Enhancements to WFQ

Since WFQ was initially proposed in 1989, many variations of WFQ have been developed with different trade-offs attempting to balance complexity, accuracy, and performance. Among the well-known WFQ variants are these four:

- Class-based WFQ assigns packets to queues based on user-defined packet classification criteria. For example, packets can be assigned to a particular queue based on the setting of the IP precedence bits. After packets are assigned to their queues, they can receive prioritized service based upon user-configured weights assigned to the different queues.
- Self-clocking Fair Queuing (SCFQ) is an enhancement to WFQ that simplifies the complexity of calculating the finish time in a corresponding GPS system. The decrease in complexity results in a larger worse-case delay and the delay increases with the number of service classes.
- Worst-case Fair Weighted Fair Queuing (WF^2Q) is an enhancement to WFQ that uses both the start and finish times of packets to achieve a more accurate simulation of a GPS system.
- Worst-case Fair Weighted Fair Queuing+ (WF^2Q+) is an enhancement to WF^2Q which implements a new virtual time function that results in lower complexity and higher accuracy.

WFQ Implementations and Applications

WFQ is deployed at the edges of the network to provide a fair distribution of bandwidth among a number of different service classes. WFQ can generally be configured to support a range of behaviors:

- WFQ can be configured to classify packets into a relatively large number of queues using a hash function that is calculated across the source/destination address pair, the source/destination UDP/TCP port numbers, and the IP ToS byte.
- WFQ can be configured to allow the system to schedule a limited number of queues that carry aggregated traffic flows. For this configuration option, the system uses QoS policy or the three low-order IP precedence bits in the IP ToS byte to assign packets to queues. Each of the queues is allocated a different percentage of output port bandwidth based on the weight that the system calculates for each of the service classes. This approach allows the system to allocate different amounts of bandwidth to each queue based on the QoS policy group or to allocate increasing amounts of bandwidth to each queue as the IP precedence increases.
- An enhanced version of WFQ, sometimes referred to as class-based WFQ, can alternately be used to schedule a limited number of queues that carry aggregated traffic flows. For this configuration option, user-defined packet classification rules assign packets to queues that are allocated a user-configured percentage of output port bandwidth. This approach allows you to determine precisely what packets are grouped in a given service class and to specify the exact amount of bandwidth allocated to each service class.

Weighted Round Robin (WRR) or Class-based Queuing (CBQ)

Weighted round robin (WRR) is the foundation for a class of queue scheduling disciplines that are designed to address the limitations of the FQ and PQ models.

- WRR addresses the limitations of the FQ model by supporting flows with significantly different bandwidth requirements. With WRR queuing, each queue can be assigned a different percentage of the output port's bandwidth.
- WRR addresses the limitations of the strict PQ model by ensuring that lower-priority queues are not denied access to buffer space and output port bandwidth. With WRR queuing, at least one packet is removed from each queue during each service round.

WRR Queuing Algorithm

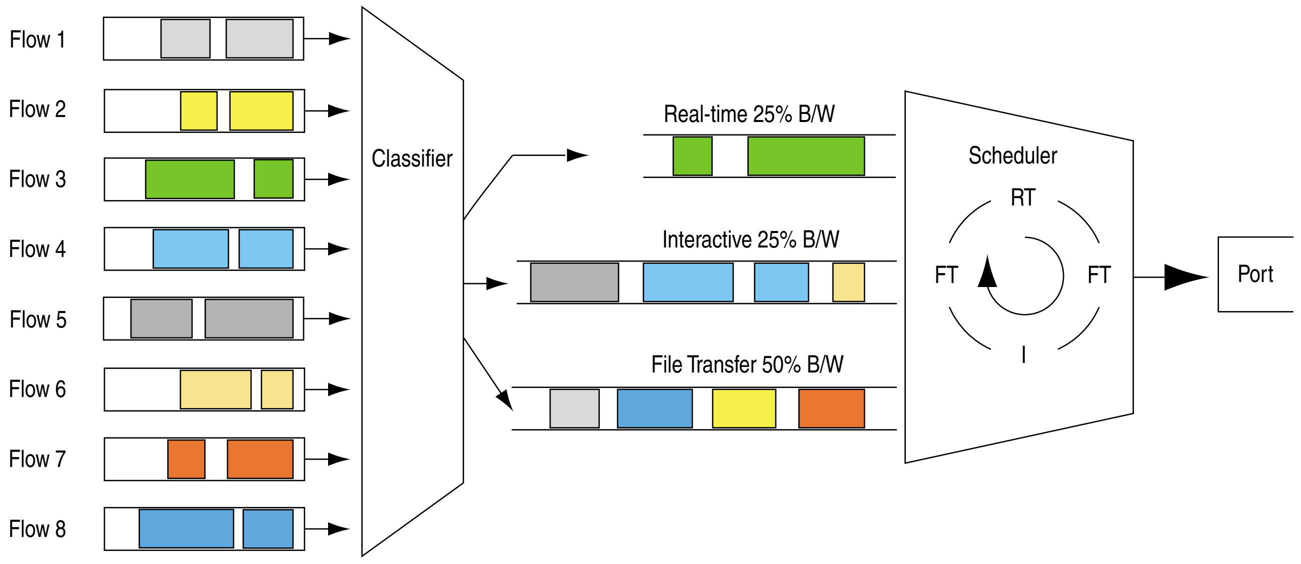
In WRR queuing, packets are first classified into various service classes (for example, real-time, interactive, and file transfer) and then assigned to a queue that is specifically dedicated to that service class. Each of the queues is serviced in a round-robin order. Similar to strict PQ and FQ, empty queues are skipped. Weighted round robin queuing is also referred to as class-based queuing (CBQ) or custom queuing.

WRR queuing supports the allocation of different amounts of bandwidth to different service class by either:

- Allowing higher-bandwidth queues to send more than a single packet each time that it visited during a service round, or
- Allowing each queue to send only a single packet each time that it is visited, but to visit higher-bandwidth queues multiple times in a single service round.

In Figure 7, the real-time traffic queue is allocated 25 percent of the output port bandwidth, the interactive traffic queue is allocated 25 percent of the output port bandwidth, and the file transfer traffic queue is allocated 50 percent of the output port bandwidth. WRR queuing supports this weighted bandwidth allocation by visiting the file transfer queue two times during each service round.

Figure 7: Weighted Round Robin (WRR) Queuing



To regulate the amount of network resources allocated to each service class, a number of parameters can be tuned to control the desired behavior of each queue:

- The amount of *delay* experienced by packets in a given queue is determined by a combination of the rate that packets are placed into the queue, the depth of the queue, the amount of traffic removed from the queue at each service round, and the number of other service classes (queues) configured on the output port.
- The amount of *jitter* experienced by packets in a given queue is determined by the variability of the delay in the queue, the variability of delay in all of the other queues, and the variability of the interval between service rounds.
- The amount of *packet loss* experienced by each queue is determined by a combination of the rate that packets are placed into the queue, the depth of the queue, the aggressiveness of the RED profiles configured for the queue, and the amount of traffic removed from the queue at each service round. The fill rate can be controlled by performing traffic conditioning at some upstream point in the network.

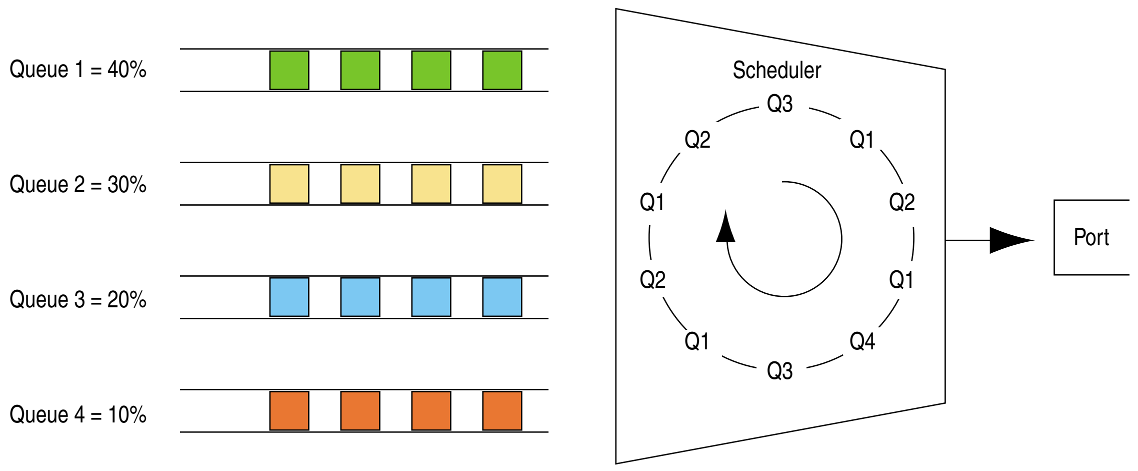
WRR Queuing Benefits and Limitations

Weighted round robin queuing includes the following benefits:

- WRR queuing can be implemented in hardware, so it can be applied to high-speed interfaces in both the core and at the edges of the network.
- WRR queuing provides coarse control over the percentage of output port bandwidth allocated to each service class.
- WRR queuing ensures that all service classes have access to at least some configured amount of network bandwidth to avoid bandwidth starvation.
- WRR queuing provides an efficient mechanism to support the delivery of differentiated service classes to a reasonable number of highly aggregated traffic flows.
- Classification of traffic by service class provides more equitable management and more stability for network applications than the use of priorities or preferences. For example, if you assign real-time traffic strict priority over file-transfer traffic, then an excessive amount of real-time traffic can eliminate all file-transfer traffic from your network. WRR queuing is based on the belief that resource reduction is a better mechanism to control congestion than resource denial. Resource denial not only blocks all traffic from lower-priority service classes but also obstructs all signaling regarding the denial of resources. As a result, TCP applications and externally clocked UDP applications are unable to correctly adapt their transmission rates to respond to the denial of network resources.

The primary limitation of weighted round-robin queuing is that it provides the correct percentage of bandwidth to each service class only if all of the packets in all of the queues are the same size or when the mean packet size is known in advance. For example, assume that you are using WRR to service four queues that are assigned the following percentages of output port bandwidth: queue 1 is allocated 40 percent, queue 2 is allocated 30 percent, queue 3 is allocated 20 percent, and queue 4 is allocated 10 percent. Assume also that all of the packets in all of the queues are 100 bytes. (See Figure 8.)

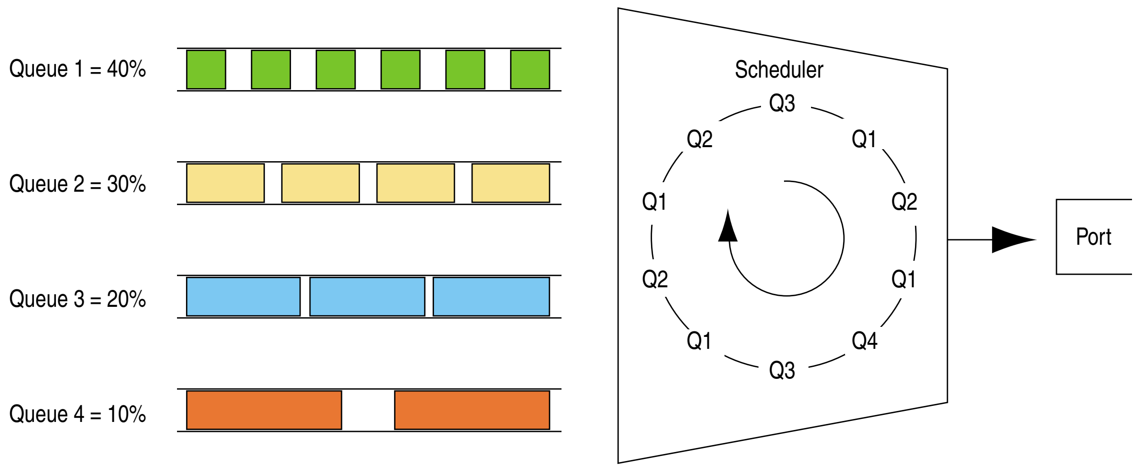
Figure 8: WRR Queuing Is Fair with Fixed-length Packets



At the end of a single service round, queue 1 transmits 4 packets (400 bytes), queue 2 transmits 3 packets (300 bytes), queue 3 transmits 2 packets (200 bytes), and queue 4 transmits one packet (100 bytes). Since a total of 1000 bytes are transmitted during the service round, queue 1 receives 40 percent of the bandwidth, queue 2 receives 30 percent of the bandwidth, queue 3 receives 20 percent of the bandwidth, and queue 4 receives 10 percent of the bandwidth. In this example, WRR queueing provides a perfect distribution of output port bandwidth. This behavior is similar to what you would expect to find in a fixed-length, cell-based ATM network, because all packets are the same size.

However, if one service class contains a larger average packet size than another service class, the service class with the larger average packet size obtains more than its configured share of output port bandwidth. Assume that the WRR scheduler is configured exactly as in the previous example. However, all of the packets in queue 1 have an average size of 100 bytes, all of the packets in queue 2 have an average size of 200 bytes, all of the packets in queue 3 have an average packet size of 300 bytes, and all of the packets in queue 4 have an average packet size of 400 bytes. (See Figure 8.)

Figure 9: WRR Queuing is Unfair with Variable-length Packets



Assuming these average packet sizes, at the end of a single service round, queue 1 transmits 4 packets (400 bytes), queue 2 transmits 3 packets (600 bytes), queue 3 transmits 2 packets (600 bytes), and queue 4 transmits 1 packet (400 bytes). Since a total of 2000 bytes are transmitted during the service round, queue 1 receives 20 percent of the bandwidth, queue 2 receives 30 percent of the bandwidth, queue 3 receives 30 percent of the bandwidth, and queue 4 receives 20 percent of the bandwidth. When faced with variable length packets, WRR queueing does not support the configured distribution of output port bandwidth.

WRR Implementations and Applications

Because the WRR scheduling discipline can be implemented in hardware, it can be deployed in both the core and at the edges of the network to arbitrate the weighted distribution of output port bandwidth among a fixed number of service classes. WRR effectively overcomes the limitations of FQ by scheduling service classes that have different bandwidth requirements. WRR also overcomes the limitations of strict PQ by ensuring that lower-priority queues are not bandwidth-starved. However, WRR's inability to support the precise allocation of bandwidth when scheduling variable-length packets is a critical limitation that needs to be addressed.

Deficit Weighted Round Robin (DWRR)

Deficit weighted round robin (DWRR) queuing was proposed by M. Shreedhar and G. Varghese in 1995. DWRR is the basis for a class of queue scheduling disciplines that are designed to address the limitations of the WRR and WFQ models.

- DWRR addresses the limitations of the WRR model by accurately supporting the weighted fair distribution of bandwidth when servicing queues that contain variable-length packets.
- DWRR addresses the limitations of the WFQ model by defining a scheduling discipline that has lower computational complexity and that can be implemented in hardware. This allows DWRR to support the arbitration of output port bandwidth on high-speed interfaces in both the core and at the edges of the network.

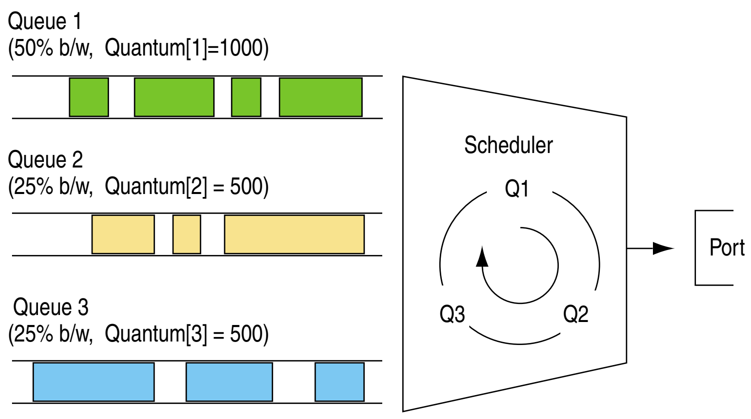
In DWRR queuing, each queue is configured with a number of parameters:

- A *weight* that defines the percentage of the output port bandwidth allocated to the queue.
- A *DeficitCounter* that specifies the total number of bytes that the queue is permitted to transmit each time that it is visited by the scheduler. The DeficitCounter allows a queue that was not permitted to transmit in the previous round because the packet at the head of the queue was larger than the value of the DeficitCounter to save transmission "credits" and use them during the next service round.
- A *quantum* of service that is proportional to the *weight* of the queue and is expressed in terms of bytes. The DeficitCounter for a queue is incremented by the quantum each time that the queue is visited by the scheduler. If quantum[i] = 2*quantum[x], then queue *i* will receive twice the bandwidth as queue *x* when both queues are active.

DWRR Algorithm

In the classic DWRR algorithm, the scheduler visits each non-empty queue and determines the number of bytes in the packet at the head of the queue. The variable DeficitCounter is incremented by the value quantum. If the size of the packet at the head of the queue is greater than the variable DeficitCounter, then the scheduler moves on to service the next queue. If the size of the packet at the head of the queue is less than or equal to the variable DeficitCouner, then the variable DeficitCounter is reduced by the number of bytes in the packet and the packet is transmitted on the output port. The scheduler continues to dequeue packets and decrement the variable DeficitCounter by the size of the transmitted packet until either the size of the packet at the head of the queue is greater than the variable DeficitCounter or the queue is empty. If the queue is empty, the value of DeficitCounter is set to zero. When this occurs, the scheduler moves on to service the next non-empty queue. (See Figure 9.)

Figure 10: Deficit Weighted Round Robin Queuing


Queue 1
(50% b/w, Quantum[1]=1000)
Queue 2
(25% b/w, Quantum[2] = 500)
Queue 3
(25% b/w, Quantum[3] = 500)
Scheduler
Q1
Q3
Q2
Port

DWRR Pseudo Code

The pseudo code in this section does not describe the operation of any specific vendor's DWRR implementation. Although each vendor's implementation will differ from this model, reviewing the examples and tracing the pseudo code will make it easier for you to understand the specific design decisions that router vendors are required to make in their implementations.

The array variable DeficitCounter is initialized to zero. In this example, the queues are numbered 1 to *n*, where *n* is the maximum number of queues on the output port:

```
FOR i = 1 to n                    /* Visit each queue index */
    DeficitCounter[i] = 0         /* Initialize DeficitCounter[i] to 0 */
ENDIF
```

The function Enqueue(i) places newly arriving packets into its correct queue and manages what is known as the ActiveList. The ActiveList is maintained to avoid examining empty queues. The ActiveList contains a list of the queue indices that contain at least one packet. Whenever a packet is placed in a previously empty queue, the index for the queue is added to the end of the ActiveList by the function InsertActiveList(i). Similarly, whenever a queue becomes empty, the index for the queue is removed from the ActiveList by the function RemoveFromActiveList(i).

```
Enqueue(i)
    i = the index of the queue that will hold the new packet
    IF (ExistsInActiveList(i) = FALSE) THEN /*IF i not in ActiveList */
        InsertActiveList(i)         /* Add i to the end of ActiveList */
        DeficitCounter[i] = 0   /*Initialize queue DeficitCounter[i] to 0*/
    ENDFOR
    Enqueue packet to Queue[i]  /* Place packet at end of queue i */
END Enqueue
```

Whenever an index is at the head of the of the ActiveList, the function Dequeue() transmits up to DeficitCounter[i] + Quantum[i] worth of bytes from queue. If, at the end of the service round Queue[i] still has packets to send, the function InsertActiveList(i) moves the index *i* to the end of the ActiveList. However, if Queue[i] is empty at the end of the service round, the DeficitCounter[i] is set to zero and the function RemoveFromActiveList(i) removes the index *i* from the ActiveList.

```
Dequeue()
    While (TRUE) DO
        IF (ActiveList is NotEmpty) THEN
            i = the index at the head of the ActiveList
            DeficitCounter[i] = DeficitCounter[i] + Quantum[i]
            WHILE (DeficitCounter[i] > 0 AND NOT Empty(Queue[i])) DO
                PacketSize = Size(Head(Queue[i]))
                IF (PacketSize <= DeficitCounter[i]) THEN
                    Transmit packet at head of Queue[i]
                    DeficitCounter[i] = DeficitCounter[i] - PacketSize
                ELSE
                    Break /*exit this while loop*/
                ENDIF
            ENDWHILE
            IF (Empty(Queue[i])) THEN
                DeficitCounter[i] = 0
                RemoveFromActiveList(i)
            ELSE
                InsertActiveList(i)
            ENDIF
        ENDIF
    ENDWHILE
END Dequeue
```

Using this pseudo code to implement a DWRR queue scheduling discipline for deployment in a production network has inherent limitations:

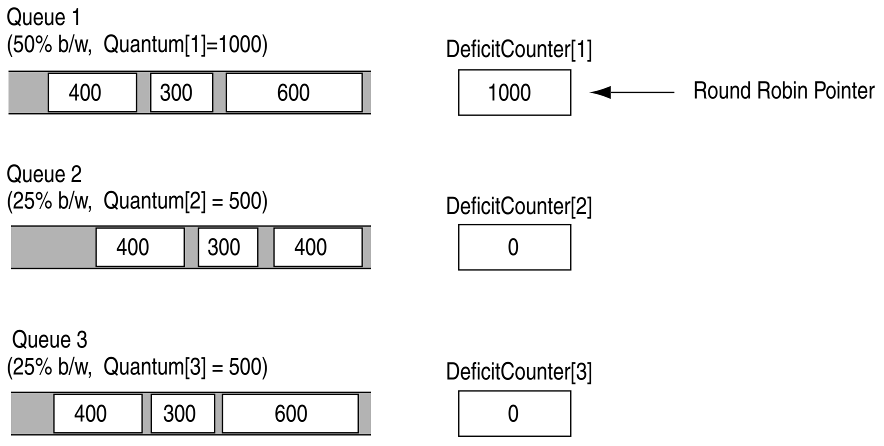
- Continuing to service multiple packets from a single queue until DeficitCounter[i] is less than the size of the packet at the head of the Queue[i] can introduce jitter, making it difficult for this implementation to support real-time traffic.
- The inability of the model to support a negative DeficitCounter[i] means that if Queue[i] does not have enough credits to transmit a packet, then the queue may experience bandwidth starvation, because it is not allowed to transmit during the current nor, perahps, subsequent service rounds until it has accumulated enough credits.

DWRR Example

For this example, assume that DWRR scheduling is enabled on an output port that is configured to support 3 queues. Queue 1 is allocated 50 percent of the bandwidth, while queue 2 and queue 3 are each allocated 25 percent of output port bandwidth.

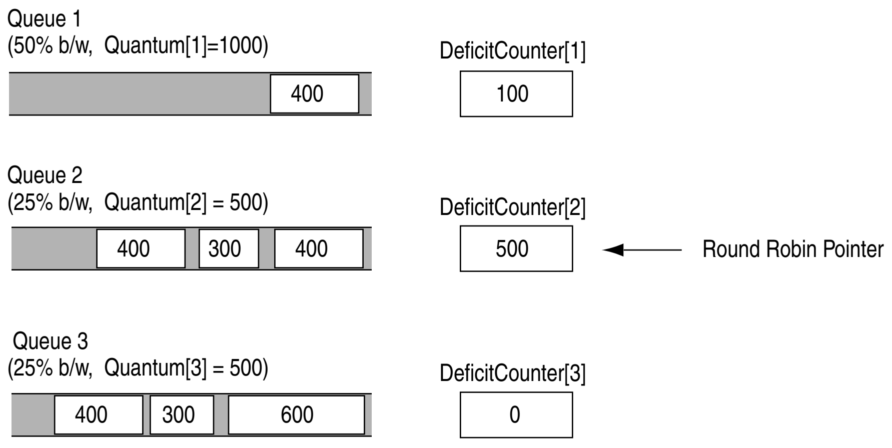
Initially, the array variable DeficitCounter is initialized to 0. Assume that the round robin pointer points to queue 1, which is at the top of the ActiveList. Before the DWRR scheduling discipline begins to service queue 1, Quantum[1] = 1000 is added to DeficitCounter[1], giving it a value of 1000. (See Figure 11.)

Figure 11: DWRR Example — Round 1 with Round Robin Pointer = 1



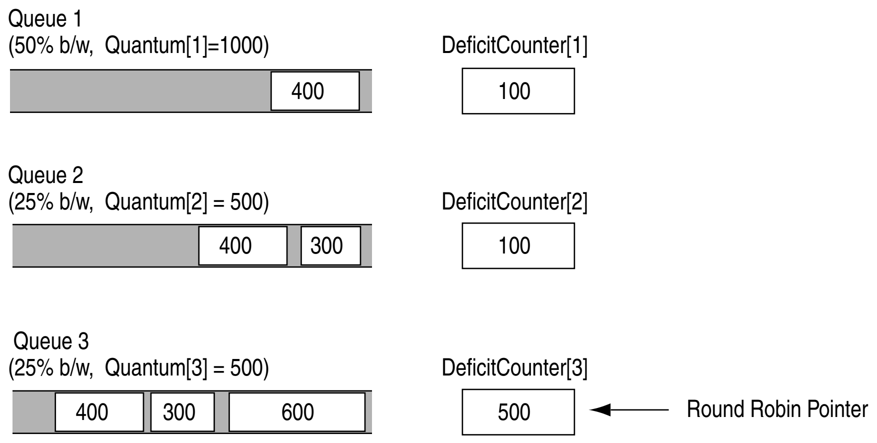
Because the 600-byte packet at the head of queue 1 is smaller than the value of DeficitCounter[1] = 1000, the 600-byte packet is transmitted. This causes the DeficitCounter[1] to be decremented by 600 bytes, resuling in a new value of 400. Now, since the 300-byte packet at the head of queue 1 is smaller than DeficitCounter[1] = 400, the 300-byte packet is also transmitted, and this causes DeficitCounter[1] to be decremented by 300 bytes, creating a new value of 100. Because the 400-byte packet at the head of queue 1 is larger than the value of DeficitCounter[1] = 100, the 400-byte packet cannot be transmitted. This causes the round robin pointer to point to queue 2, which is now at the top of the ActiveList. (See Figure 12.)

Figure 12: DWRR Example — Round 1 with Round Robin Pointer = 2



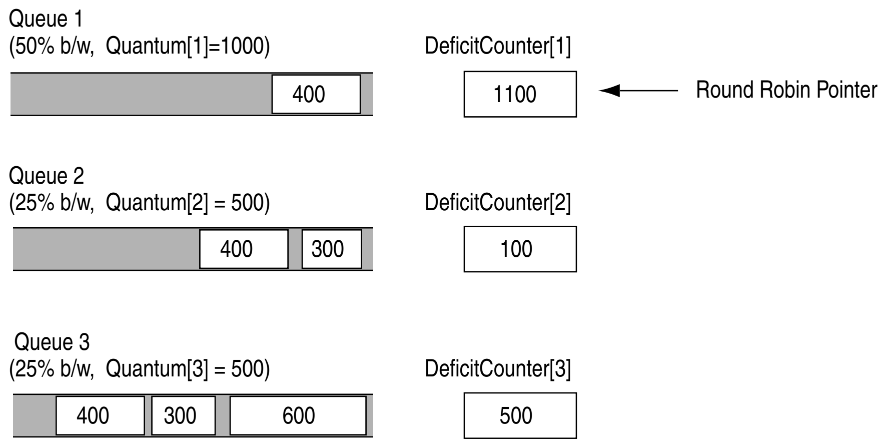
Before the DWRR scheduling discipline starts to service queue 2, Quantum[2] = 500 is added to DeficitCounter[2] giving it a value of 500. Since the 400-byte packet at the head of queue 2 is smaller than the value of DeficitCounter[2] = 500, the 400-byte packet is transmitted. This causes DeficitCounter[2] be decremented by 400 bytes and thus have a new value of 100. Because the 300-byte packet at the head of queue 2 is larger than the value of DeficitCounter[2] = 100, the 300-byte packet cannot be transmitted. This causes the round robin pointer to point to queue 3, which is now at the top of the ActiveList. (See Figure 13.)

Figure 13: DWRR Example — Round 1 with Round Robin Pointer = 3



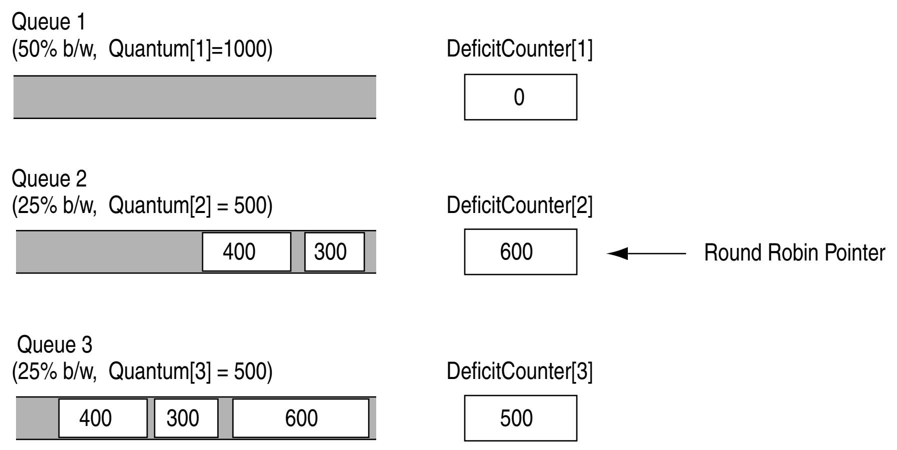
Before the DWRR scheduling discipline starts to service queue 3, Quantum[3] = 500 is added to DeficitCounter[3], giving it a value of 500. Since the 600-byte packet at the head of queue 2 is larger than the value of DeficitCounter[3] = 500, the 600-byte packet cannot be transmitted. This causes the round robin pointer to point to queue 1, which is now at the top of the ActiveList. (See Figure 14.)

Figure 14: DWRR Example — Round 2 with Round Robin Pointer = 1



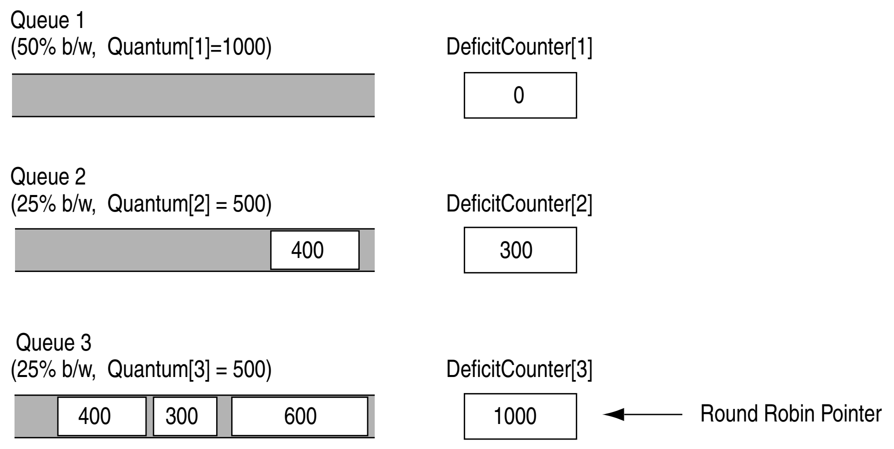
Before the DWRR scheduling discipline starts to service queue 1, Quantum[1] = 1000 is added to DeficitCounter[1] giving it a value of 1100. Since the 400-byte packet at the head of queue 1 is smaller than the value of DeficitCounter[1] = 1100, the 400-byte packet is transmitted. This causes DeficitCounter[1] be decremented by 400 bytes and have a new value of 700. Now queue 1 is empty. This causes DeficitCounter[1] to be set to zero, queue 1 to be removed from the ActiveList, and the round robin pointer to point to queue 2, which is now at the top of the ActiveList. (See Figure 15.)

Figure 15: DWRR Example — Round 2 with Round Robin Pointer = 2



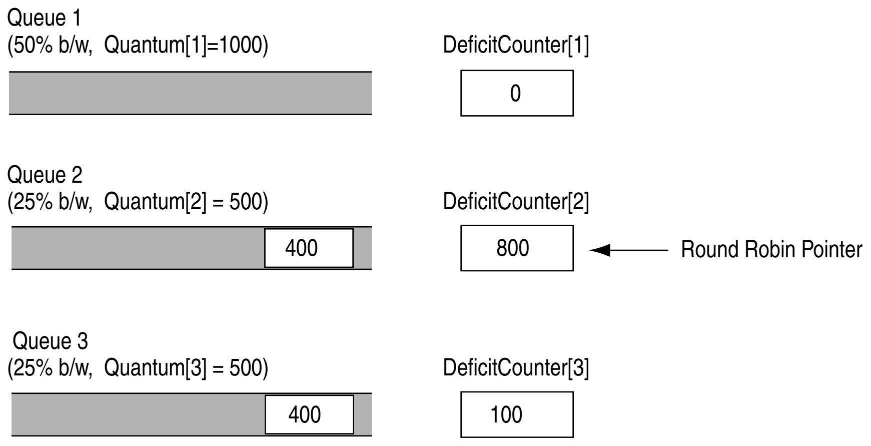
Before the DWRR scheduling discipline starts to service queue 2, Quantum[2] = 500 is added to DeficitCounter[2], giving it a value of 600. Since the 300-byte packet at the head of queue 2 is smaller than the value of DeficitCounter[2] = 600, the 300-byte packet is transmitted. This causes DeficitCounter[2] to be decremented by 300 bytes, to a new value of 300. Since the 400-byte packet at the head of queue 2 is larger than the value of DeficitCounter[2] = 300, the 400-byte packet cannot be transmitted. This causes the round robin pointer to point to queue 3, which is now at the top of the ActiveList. (See Figure 16.)

Figure 16: DWRR Example — Round 2 with Round Robin Pointer = 3



Before the DWRR scheduling discipline starts to service queue 3, Quantum[3] = 500 is added to DeficitCounter[3] giving it a value of 1000. Since the 600-byte packet at the head of queue 3 is smaller than the value of DeficitCounter[3] = 1000, the 600-byte packet is transmitted. This causes DeficitCounter[3] be decremented by 600 bytes and have a new value of 400. Since the 300-byte packet at the head of queue 3 is smaller than the value of DeficitCounter[3] = 400, the 300-byte packet is transmitted, and this causes DeficitCounter[3] to be decremented by 300 bytes, to a new value of 100. Since the 400-byte packet at the head of queue 3 is larger than the value of DeficitCounter[3] = 100, the 400-byte packet cannot be transmitted. This causes the round robin pointer to point to queue 2, which is now at the top of the ActiveList. (Figure 17.)

Figure 17: DWRR Example — Round 3 with Round Robin Pointer = 2



Before the DWRR scheduling discipline starts to service queue 2, Quantum[2] = 500 is added to DeficitCounter[2], giving it a value of 800. At this point, the DWRR scheduling discipline continues to service queues by providing the user-configured percentage of output port bandwidth to each service class.

DWRR Benefits and Limitations

The benefits of DWRR queuing are that it:

- Provides protection among different flows, so that a poorly behaved service class in one queue cannot impact the performance provided to other service classes assigned to other queues on the same output port;
- Overcomes the limitations of WRR by providing precise controls over the percentage of output port bandwidth allocated to each service class when forwarding variable-length packets;
- Overcomes the limitations of strict PQ by ensuring that all service classes have access to at least some configured amount of output port bandwidth to avoid bandwidth starvation; and
- Implements a relatively simple and inexpensive algorithm, from a computational perspective, that does not require the maintenance of a significant amount of per-service class state.

As with other models, DWRR queuing has limitations:

- Highly aggregated service classes mean that a misbehaving flow within a service class can impact the performance of other flows within the same service class. However, in the core of a large IP network, routers are required to schedule aggregate flows, because the large number of individual flows makes it impractical to support per-flow queue scheduling disciplines.

- DWRR does not provide end-to-end delay guarantees as precise as other queue scheduling disciplines do.
- DWRR may not be as accurate as other queue scheduling disciplines. However, over high-speed links, the accuracy of bandwidth allocation is not as critical as over low-speed links.

DWRR Implementations and Applications

Because the DWRR queue scheduling discipline can be implemented in hardware, it can be deployed in both the core and at the edges of the network to arbitrate the weighted distribution of output port bandwidth among a fixed number of service classes. DWRR provides all of the benefits of WRR, while also addressing the limitations WRR by supporting the accurate allocation of bandwidth when scheduling variable-length packets.

Conclusion

In this paper, we described the operation, benefits, and limitations of a number of classic queue scheduling disciplines:

- First-in, first-out (FIFO)
- Priority queuing (PQ)
- Fair queuing (FQ)
- Weighted fair queuing (WFQ)
- Weighted round robin (WRR) or class-based queuing (CBQ)
- Deficit weighted round robin (DWRR)

The very nature of large IP production networks requires that router vendors offer a combination of these approaches if they are to provide a solution that allows you to accurately arbitrate service-class access to output port bandwidth during periods of congestion. Each vendor's implementation seeks to find the correct balance between performance, accuracy, and simplicity.

For example, one possible solution combines rate-controlled PQ with DWRR queuing. In this hardware-based approach, high priority queues are serviced strictly before low-priority queues, as long as high-priority queues do not exceed their allocated bandwidth. If a high-priority queue is rate-controlled, it cannot exceed its allocated bandwidth, so it is impossible for it to starve low-priority queues. Consequently, a high-priority queue is treated preferentially only as long as it stays within its allocated bandwidth. Within a priority, queues can be serviced in a DWRR manner to provide accurate control over the percentage of output port bandwidth allocated to each queue within the priority. This combination of rate-controlled PQ and DWRR is computationally inexpensive, can be implemented in hardware, and allows high-priority, delay-sensitive traffic to be serviced quickly while avoiding bandwidth starvation for other types of low-priority traffic.

References

Textbooks

Croll, Alistair and Packman, Eric. *Managing Bandwidth: Deploying QoS in Enterprise Networks.* Prentice Hall PTR, January 2000. (ISBN 0130113913)

Ferguson, Paul and Huston, Geoff. *Quality of Service: Delivering QoS on the Internet and in Corporate Networks.* John Wiley & Sons, January 1998. (ISBN 0471243582)

Huitema, Christian. *Routing in the Internet.* Prentice Hall PTR, January 2000. (ISBN 0130226475)

Huston, Geoff. *Internet Performance Survival Guide: QoS Strategies for Multiservice Networks.* John Wiley & Sons, February 2000. (ISBN 0471378089)

Kilkki, Kalevi. *Differentiated Services for the Internet*. New Riders Publishing, June 1999. (ISBN 1578701325)

Partridge, Craig. *Gigabit Networking.* Addison-Wesley Pub Co., January 1994. (ISBN 0201563339)

Stevens, W. Richard. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley Publishing Co., January 1994. (ISBN 0201633469)

Wang, Zheng. *Internet QoS: Architectures and Mechanisms for Quality of Service.* Morgan Kaufmann Publishers, March 2001. (ISBN: 1558606084)

Technical Papers

Bennett, J. and Zhang, H. "Hierarchical Packet Fair Queueing Algorithms." *Proc. ACM SIGCOMM '96 (*August 1996): 675–689.

Bennet, J. and Zhang, H. "WF2Q: Worst-case Fair Weighted Fair Queueing." Proceedings of *IEEE INFOCOM ´96* (March 1996): 120–128.

Demers, A., Keshav, S., and Shenker, S. "Analysis and Simulation of a Fair-queueing Algorithm." Proc. ACM SIGCOMM '89 (September 1989): 1–12.

Nagle, J. "On Packet Switches with Infinite Storage." (RFC 970) *IEEE Transactions on Communications*, vol. 35, no. 4 (April 1987): 435–438.

Parekh, A. "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks." Doctoral dissertation, Massachusetts Institute of Technology, February 1992.

Shreedhar, M. and Varghese, G. "Efficient Fair Queueing Using Deficit Round Robin." *Proc. ACM SIGCOMM '95*, Vol. 25, No. 4 (October 1995):231–242.

Zhang, Lixia. "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks." *Proc. ACM SIGCOMM '90* (1990): 19–29.

Seminars

Metz, Chris. "A Survey of Advanced Internet protocols," Next-Generation Networks Conference, Washington, D.C., 2000.

Perkins, Drew. "Quality of Service for the Integrated Services Internet, Next-Generation Networks Conference, Washington, D.C., 1997.

Web Sites

Bonaventure, Olivie. "Packet Level Traffic Control Mechanisms." 2000. http://enligne.infonet.fundp.ac.be/coursenligne/cours/00-01/INFO2231/INFO2231-2.big/index.htm