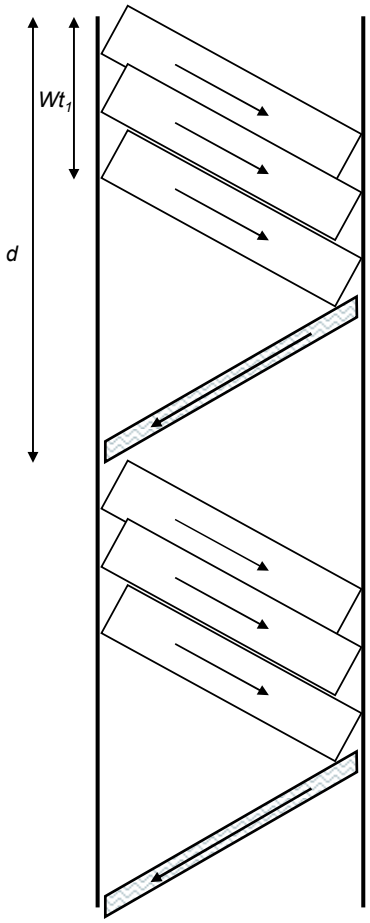
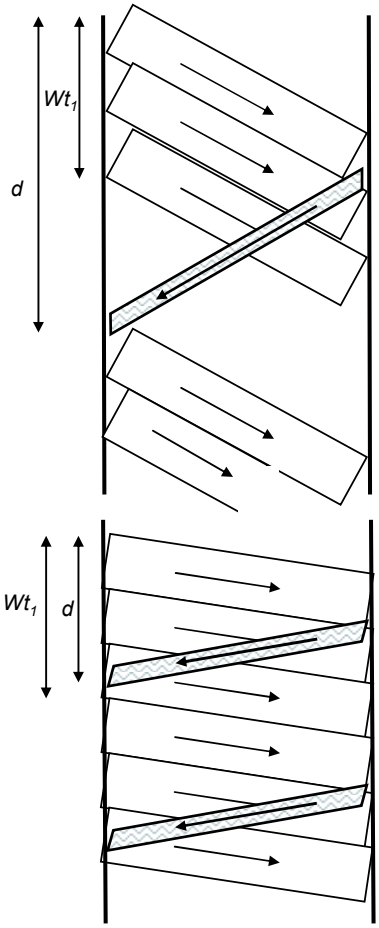
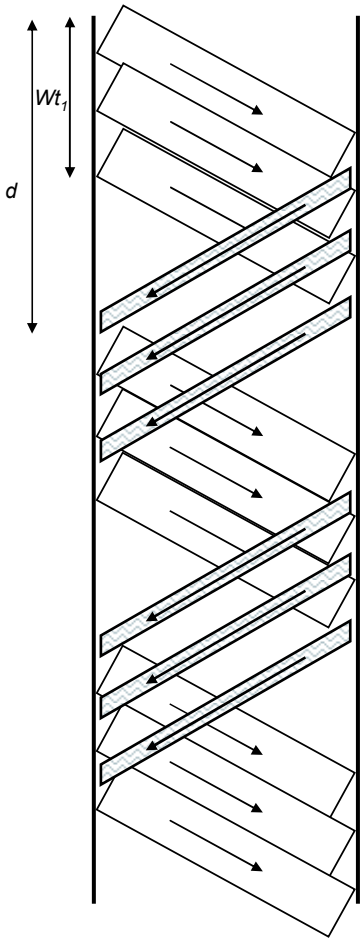
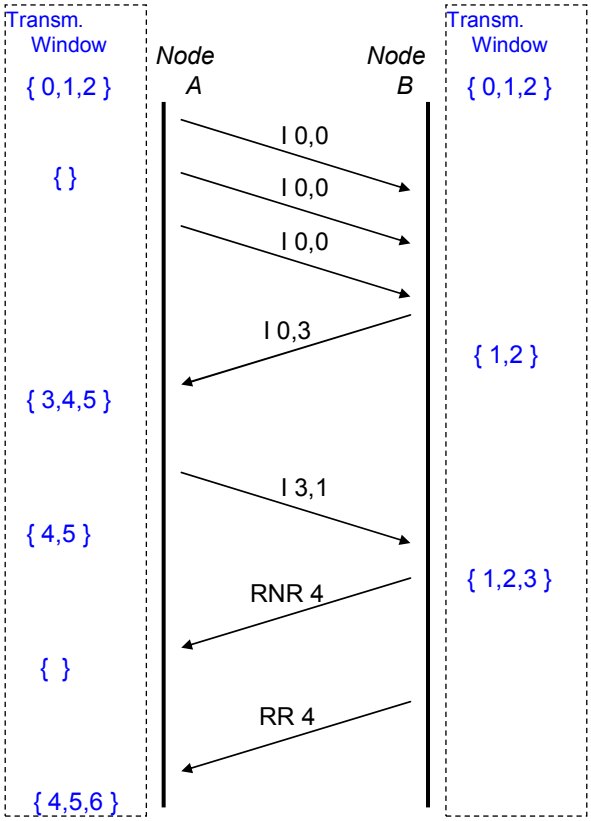

Wt1
d


Wt1
d
Wt1
d


Wt1
d

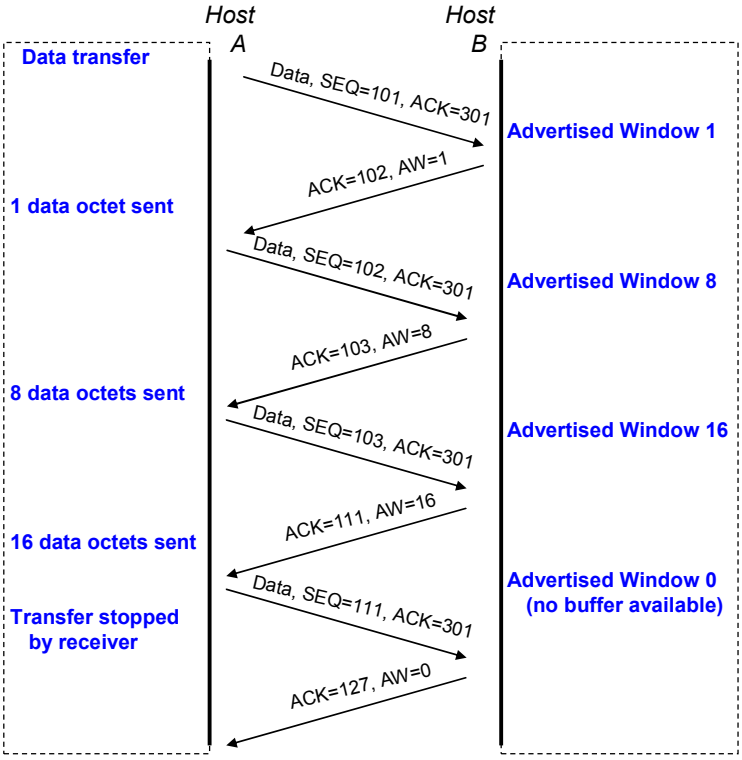
HDLC Window


Transm. Window
{ 0,1,2 }
{ }
{ 3,4,5 }
{ 4,5 }
{ }
{ 4,5,6 }
Node A
Node B
I 0,0
I 0,0
I 0,0
I 0,3
I 3,1
RNR 4
RR 4
Transm. Window
{ 0,1,2 }
{ 1,2 }
{ 1,2,3 }

TRANSPORT LEVEL FLOW CONTROL

TCP SLIDING WINDOW

TCP Window



TCP Slow start:
actual window W = min (AW, CW)
Receiver advertised windows AW, Congestion window CW
CW = 1
CW incremented by one segment each time ACK is received (exponential growth of CW)

TCP Congestion avoidance:
SST - slow start threshold
1. initially CW = 1, SST = 65 535 octets
2. congestion detected (ACK timeout or duplicate ACKs):
   SST = 0.5 * current window W (but at least 2 segments)
3. if (ACK timeout) then CW = 1 else CW = CW / 2
   if (CW < SST) then perform *TCP Slow start*
   else perfom *TCP Congestion avoidance:*
   CW incremented by [segsize*segsize / SST] each time ACK is received

The increase in CW should be at most one segment each round-trip time (regardless how many ACKs are received in that RTT), whereas *Slow start* increments CW by the number of ACKs received in a round-trip time.

TCP Fast Retransmit
TCP may generate an immediate acknowledgment (a *duplicate ACK*) when an out-of-order segment is received. This duplicate ACK should not be delayed. The purpose of this duplicate ACK is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected. Since TCP does not know whether a duplicate ACK is caused by a lost
segment or just a reordering of segments, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If *three or more duplicate ACKs are received in a row*, it is a strong indication that a *segment has been lost.* TCP then performs a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire.

TCP Fast recovery
After fast retransmit sends what appears to be the missing segment, congestion avoidance, but not slow start is performed. This is the fast recovery algorithm. It is an improvement that allows high throughput under moderate congestion, especially for large windows.

NETWORK NODE LEVEL FLOW CONTROL - BUFFER MANAGEMENT

CHANNEL QUEUE LIMIT (CQL)
Incoming packets (cells) are distinguished based on the output queue (channel) they must be placed to. Some threshold t_i (fixed or dynamic) is defined for each queue i. Packets (cells) beyond this threshold are discarded.

B - total buffer size, N - number of channels

- Complete partitioning CP: $t_i < B$; $\Sigma\, t_i = B$
- Complete sharing CS: $t_i = B$
- Sharing with minimum allocation (SMA):
  certain buffer space t_{iMIN} is reserved for each channel : $\Sigma\, t_{iMIN} < B$
  rest of buffer space is completely shared among all channels.

Dynamic schemes

- Pushout: if the buffer is full, an arriving packet (cell) will make room for itself by pushing out the packet at the end of the longest queue. Pushout allows dynamic complete sharing.
- Selective pushout: it is allowed to push out only taged packet (cell).
- Dynamic thresholding: tresholds can vary according to the overall buffer occupancy (dynamic SMA)

STRUCTURED BUFFER POOL (SBP)
SBP distinguishes incoming packets based on the “hop count”. Buffer is arranged in levels from 0 to H (H is maximum number of hops which can packet take on the way to destination). Level i is reserved for packets that have covered at least i hops.

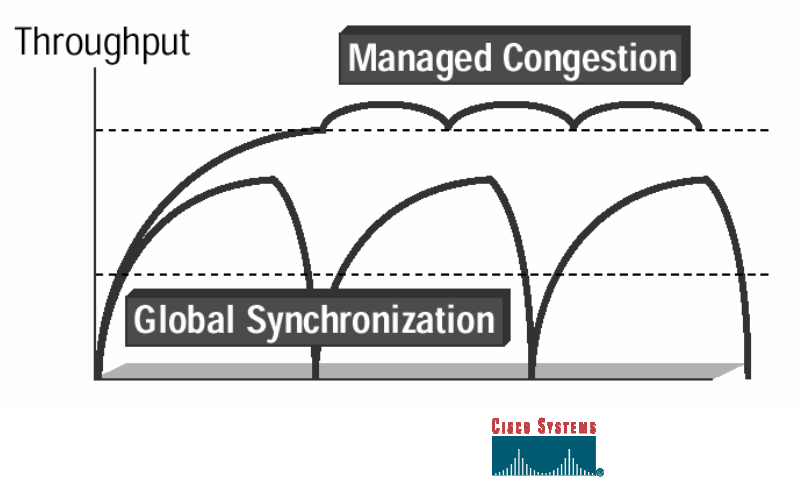
EARLY PACKET DISCARD (EPD)
Node dumps the cells from packets it has not yet started to ship.

PARTIAL PACKET DISCARD (PPD)
Node dumps the cells from packets it has started to ship.

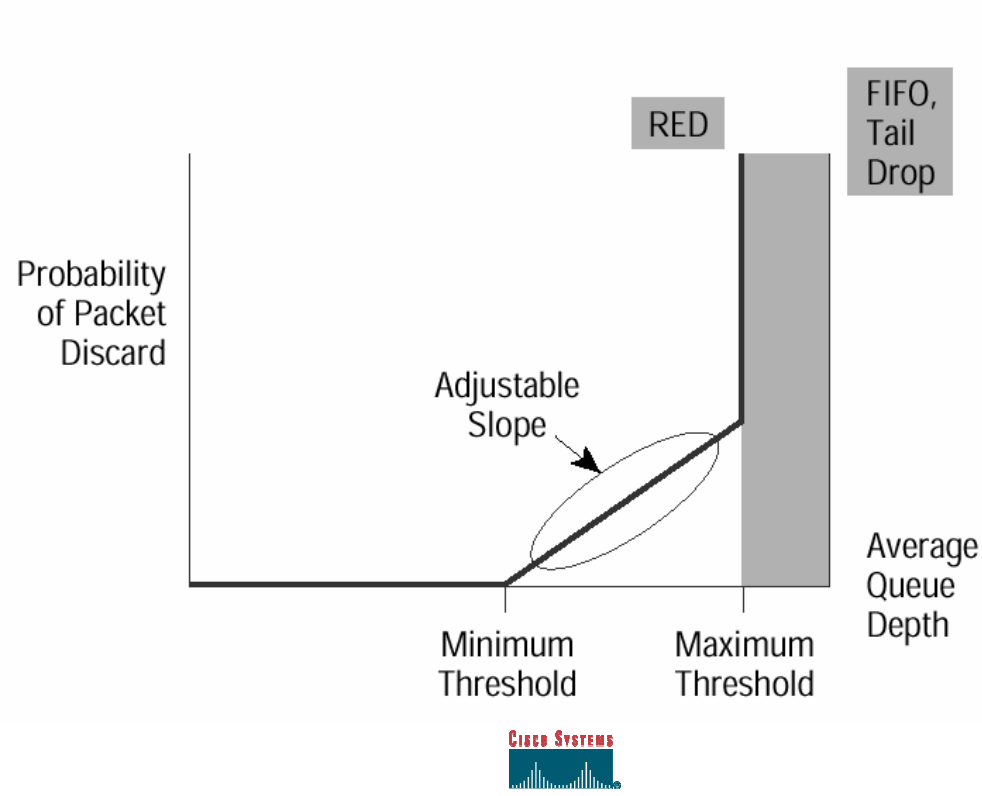
RANDOM EARLY DISCARD
With multiple TCP sources, the dropping packets uniformly from all sources causes all of them to back off and then begin retransmission all at once. This scenario leads to waves of congestion, also referred to as “global synchronization.”
RED solves this problem by dropping packets selectively from specific TCP flows so that only a few of the TCP senders back off and retransmit.

RANDOM EARLY DISCARD

With multiple TCP sources, the dropping packets uniformly from all sources causes all of them to back off and then begin retransmission all at once. This scenario leads to waves of congestion, also referred to as “global synchronization.”



RED solves this problem by dropping packets selectively from specific TCP flows so that only a few of the TCP senders back off and retransmit.



TRAFFIC POLICING & TRAFFIC SHAPING

Leaky Bucket

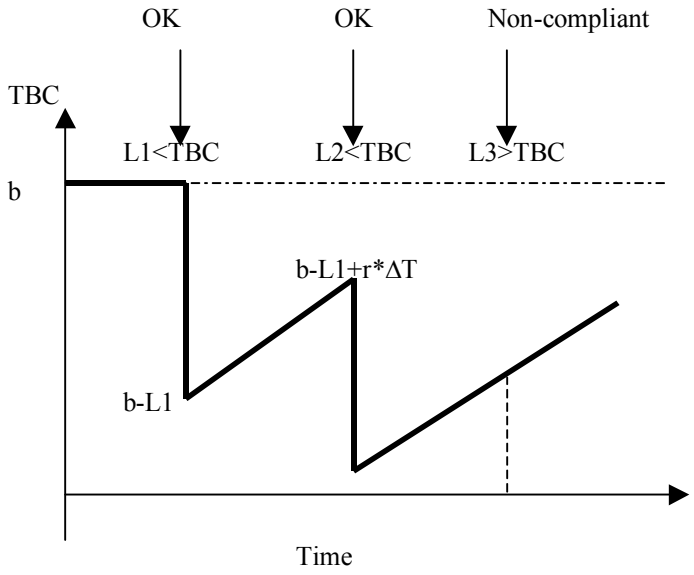
Token Bucket

(3rd Generation Partnership Project : QoS Concept and Architecture 3GPP TS 23.107)

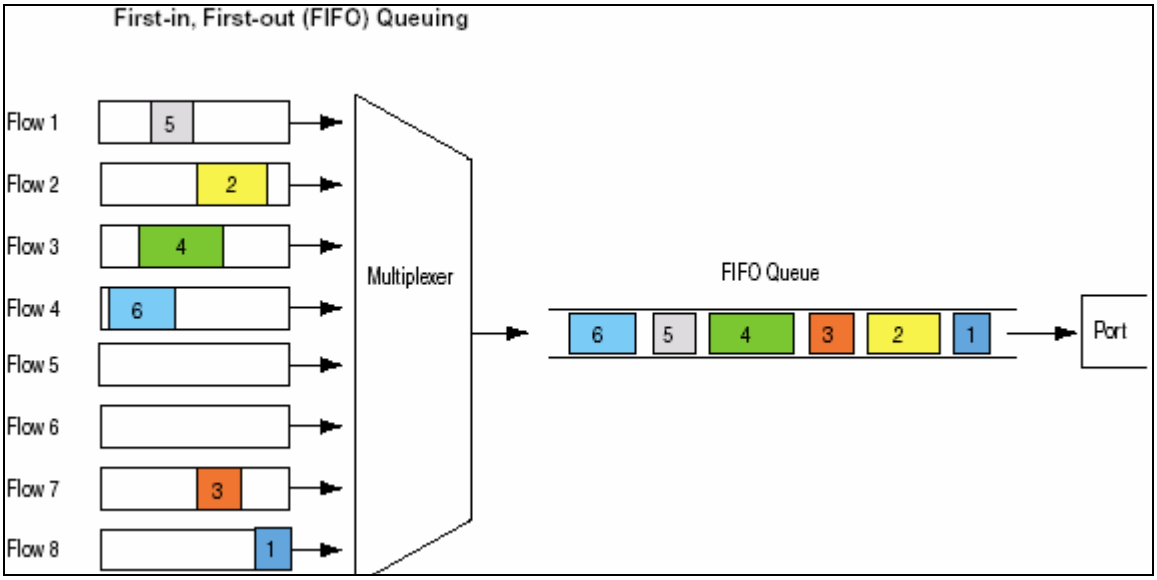
Here, "tokens" represents the allowed data volume, for example in byte. "Tokens" are given at a constant "token rate" by a traffic contract, are stored temporarily in a "token bucket", and are consumed by accepting the packet.

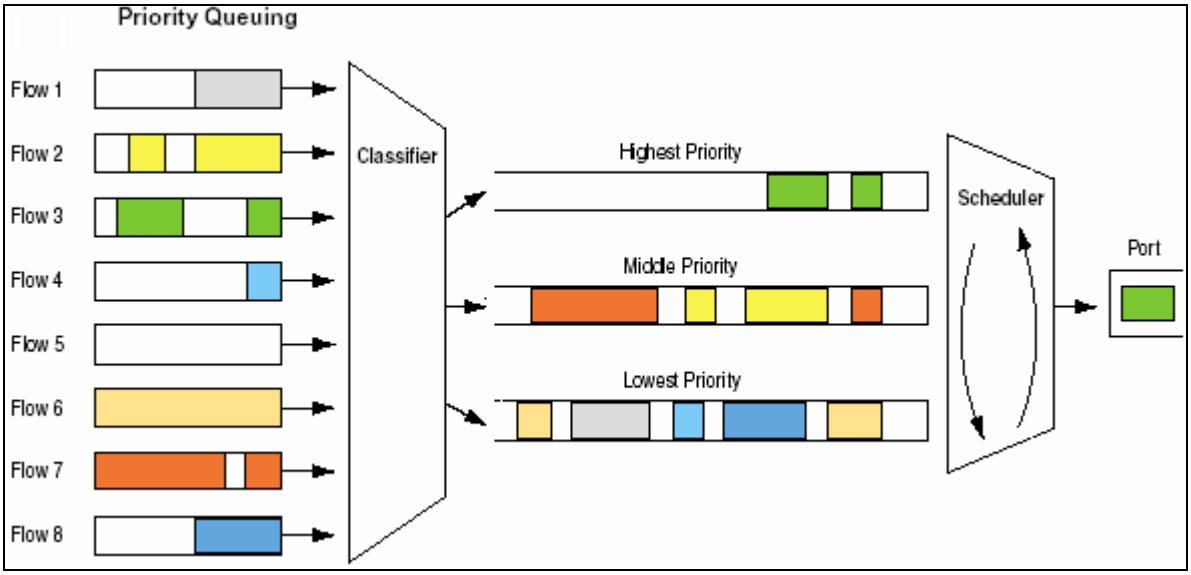
- ***r***: token rate, (corresponds to the monitored Maximum bitrate/Guaranteed bitrate).
- ***b***: bucket size, (the upper bound of TBC, corresponds to bounded burst size).
- ***TBC***(Token bucket counter): the number of given/remained tokens at any time. ***TBC*** is usually increased by "***r***" in each small time unit. However, TBC has upper bound "***b***" and the value of ***TBC*** shall never exceed "***b***".

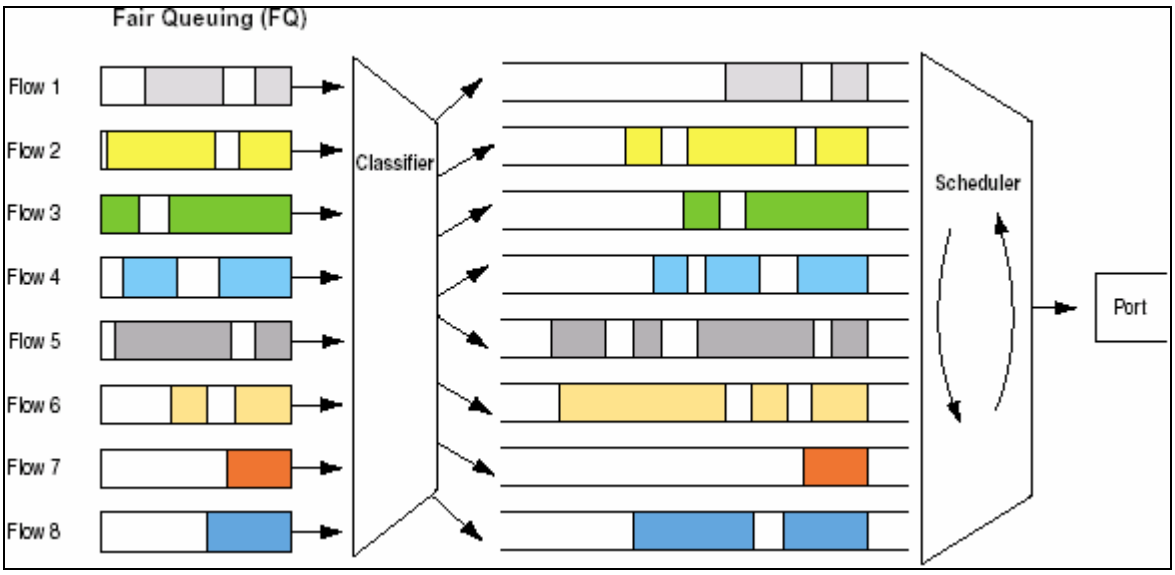
When a packet ***#i*** with length ***Li*** arrives, the receiver checks the current ***TBC***. If the ***TBC*** value is equal to or larger than ***Li***, the packet arrival is judged compliant, i.e., the traffic is conformant. At this moment tokens corresponding to the packet length is consumed, and ***TBC*** value decreases by ***Li***.
When a packet **#j** with length ***Lj*** arrives, if ***TBC*** is less than ***Lj***, the packet arrival is non-compliant. In this case, the value of ***TBC*** is not updated

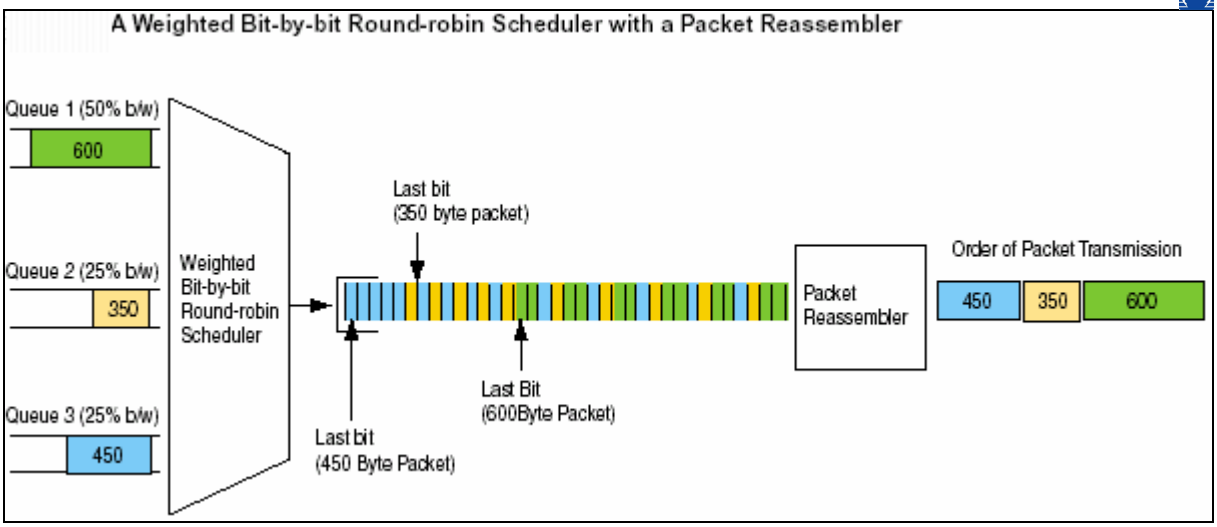


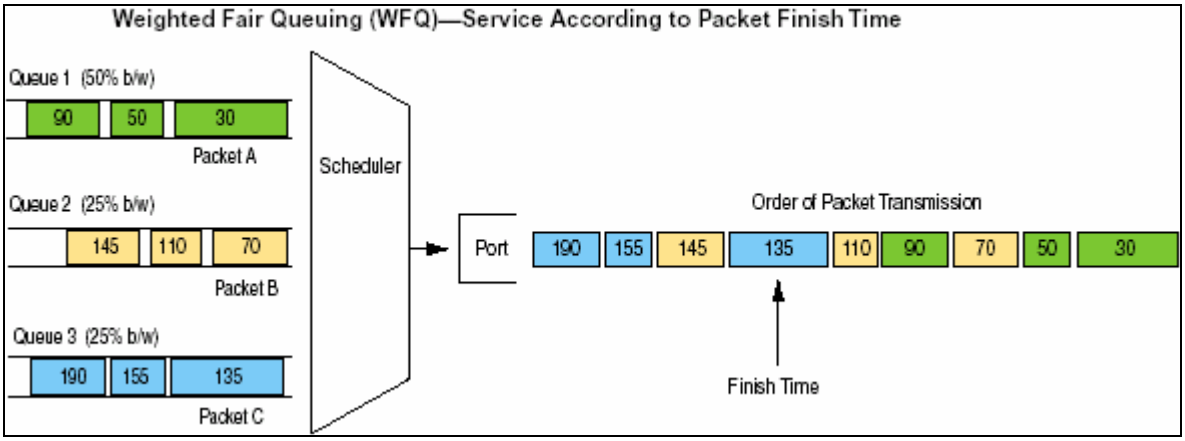
SCHEDULING AND QUEUING

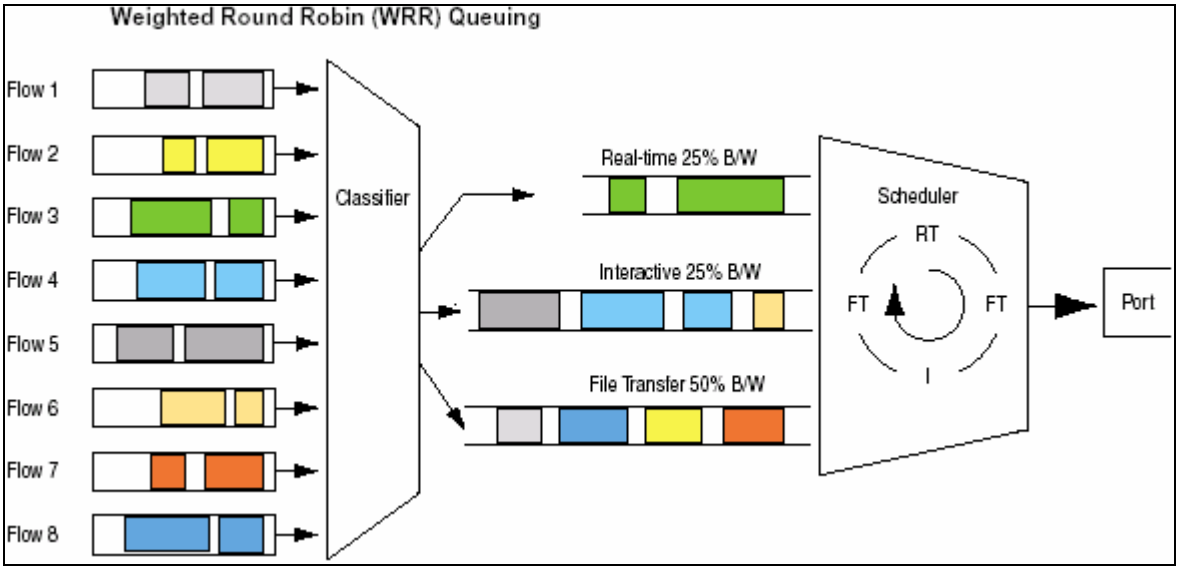







A Weighted Bit-by-bit Round-robin Scheduler with a Packet Reassembler
Queue 1 (50% b/w)
600
Queue 2 (25% b/w)
350
Queue 3 (25% b/w)
450
Weighted Bit-by-bit Round-robin Scheduler
Last bit (350 byte packet)
Last bit (450 Byte Packet)
Last Bit (600Byte Packet)
Packet Reassembler
Order of Packet Transmission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


Weighted Fair Queuing (WFQ)—Service According to Packet Finish Time
Queue 1 (50% b/w)
90
50
30
Packet A
Queue 2 (25% b/w)
145
110
70
Packet B
Queue 3 (25% b/w)
190
155
135
Packet C
Scheduler
Port
Order of Packet Transmission
190
155
145
135
110
90
70
50
30
Finish Time


Weighted Round Robin (WRR) Queuing
Flow 1
Flow 2
Flow 3
Flow 4
Flow 5
Flow 6
Flow 7
Flow 8
Classifier
Real-time 25% B/W
Interactive 25% B/W
File Transfer 50% B/W
Scheduler
RT
FT
FT
I
Port