Differentiated Service Queuing Disciplines in NS-3

Robert Chang, Mahdi Rahimi, and Vahab Pournaghshband

Advanced Network and Security Research Laboratory
California State University, Northridge
Northridge, California, USA
{robert.i.chang, mahdi.r.rahimi}@ieee.org
vahab@csun.edu

***Abstract*—Network Simulator 3 (`ns-3`) is a powerful tool for modeling the behavior of computer networks using simulation. We have developed three well known differentiated service packet queuing methods: strict priority queuing, weighted fair queuing, and weighted round robin queuing, in the simulation framework. In this paper, we present the implementation details of the three modules as well as their usage. By implementing these modules in `ns-3` and demonstrating their use, we intend to facilitate further research and experimentation with our contributions. We believe that our work will be utilized in solving outstanding problems that would have been impractical to investigate without our modules. Lastly, through validation, we confirm that our introduced modules simulate these queuing methods correctly.**

Keywords–strict priority queuing; weighted fair queuing; weighted round robin; differentiated service; ns-3

I. Introduction

We are presenting three new modules for three scheduling strategies: strict priority queuing, weighted fair queuing, and weighted round robin. These queuing methods offer differentiated service to network traffic flows, optimizing performance based on administrative configurations.

The network simulator 3 (`ns-3`) [1] is a popular and valuable research tool which can be used to simulate systems and evaluate network protocols. `ns-3` organizes components logically into modules. The official modules included by default are able to create basic simulated networks using common protocols, and users can add additional components by creating specialized modules. This has been used to add a leaky bucket scheduler [2] and to add and evaluate a DiffServ framework implementation [3].

DiffServ is a network architecture that provides a way to differentiate and manage network flows. A DiffServ network can give priority to real-time applications, such as Voice over IP, to ensure acceptable performance, or prevent malfunctioning and malicious applications from occupying all of the bandwidth and starving other communication. Two of the main components of DiffServ are classification and scheduling. DiffServ networks classify the packets in a network flow to determine what kind of priority or service to provide and schedule packets according to their classification. Differentiated service queuing disciplines, such as those described in this paper, are responsible for executing the flow controls required by DiffServ networks.

This paper is organized as follows: first, a brief overview of the theoretical background behind each of our modules is presented in Section II. In Section III, we overview existing simulation tools for differentiated service queuing. Section III describes our design choices and implementation details. Section IV showcases experiments using our modules and presents an analysis of the results to validate their correctness by comparing the observed behavior to analytically-derived expectations. In Section V, we provide instructions to configure these modules in an `ns-3` simulation, and finally. we consider future work in Section VI.

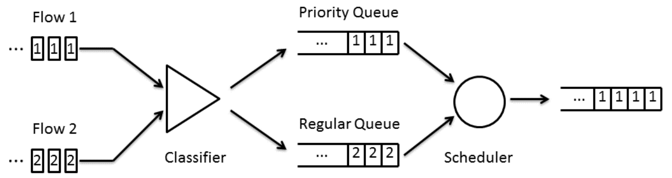


Figure 1. A strict priority queue

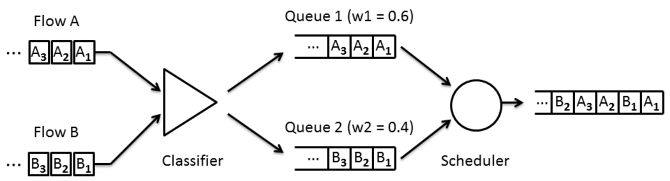
II. Background

A. Strict Priority Queuing

Strict priority queuing (SPQ) [4] classifies network packets as either priority or regular traffic and ensures that priority traffic will always be served before low priority. Priority packets and regular packets are filtered into separate FIFO queues, the priority queue must be completely empty before the regular queue will be served. The advantage of this method is that high priority packets are guaranteed delivery so long as their inflow does not exceed the transmission rate on the network. The potential disadvantage is a high proportion of priority traffic will cause regular traffic to suffer extreme performance degradation [4]. Figure 1 gives an example of strict priority queuing; packets from flow 2 cannot be sent until the priority queue is completely emptied of packets from flow 1.

B. Weighted Fair Queuing

Weighted fair queuing (WFQ) [5] offers a more balanced approach than SPQ. Instead of giving certain traffic flows complete precedence over others, WFQ divides traffic flows into two or more classes and gives a proportion of the available bandwidth to each class based on the idealized Generalized Processor Sharing (GPS) model [6].



Packet	Finish Time
A_1	35
A_2	60
A_3	80
B_1	50
B_2	90
B_3	110

Figure 2. A weighted fair queue

In GPS, each queue i is assigned a class of traffic and weight w_i. At any given time the weights corresponding to nonempty queues w_j are normalized to determine the portion of bandwidth allocated to the queue as shown in (1).

$$w_i^* = \frac{w_i}{\sum w_j} \tag{1}$$

w_i^* is between zero and one and is equal to the share of the total bandwidth allocated to queue i. For any t seconds on a link capable of sending b bits per second, each nonempty queue sends $b * t * w_i^*$ bits.

WFQ approximates GPS by calculating the order in which the last bit of each packet would be sent by a GPS scheduler and dequeues packets in this order [7]. The order of the last bits is determined by calculating the virtual finish time of each packet. WFQ assigns each packet a start time and a finish time, which correspond to the virtual times at which the first and last bits of the packet, respectively, are served in GPS. When the kth packet of flow i, denoted by P_i^k, arrives at the queue, its start time and finish time are determined by (2) and (3).

$$S_i^k = max(F_i^{k-1}, V(A_i^k)) \tag{2}$$

$$F_i^k = S_i^k + \frac{L_i^k}{w_i} \tag{3}$$

where $F_i^0 = 0$, A_i^k is the actual arrival time of packet P_i^k, L_i^k is the length of P_i^k, and w_i is the weight of flow i. Here $V(t)$ is the virtual time at real time t to denote the current round of services in GPS and is defined in (4).

$$\frac{dV(t)}{dt} = \frac{c}{\sum\limits_{i \in B_{\langle t \rangle}} w_i} \tag{4}$$

where $V(0) = 0$, c is the link capacity, and $B_{\langle t \rangle}$ is the set of backlogged connections at time t under the GPS reference system. WFQ then chooses which packet to dequeue based on the minimal virtual finish time. Figure 2 gives an example of weighted fair queuing; packets are sent in the order determined by their virtual finish times.

C. Weighted Round Robin

Weighted round robin (WRR) queuing is a round robin scheduling algorithm that approximates GPS in a less computationally intensive way than WFQ. Every round each nonempty queue transmits an amount of packets proportional to its weight. If all packets are of uniform size, each class of traffic is provided a fraction of bandwidth exactly equal to its assigned weight. In the more general case of IP networks with variable size packets, the weight factors must be normalized using the mean packet size. Normalized weights are then used to determine the number of packets serviced from each queue. If w_i is the assigned weight for a class and L_i is the mean packet size, the normalized weight of each queue is given by (5).

$$w_i^* = \frac{w_i}{L_i} \tag{5}$$

Then the smallest normalized weight, w_{min}^*, is used to calculate the number packets sent from queue i each round as shown in (6) [8].

$$\left\lceil \frac{w_i^*}{w_{min}^*} \right\rceil \tag{6}$$

WRR has a processing complexity of $O(1)$, making it useful for high speed interfaces on a network. The primary limitation of WRR is that it only provides the correct proportion of bandwidth to each service class if all packets are of uniform size or the mean packet size is known in advance, which is very uncommon in IP networks. To ensure that WRR can emulate GPS correctly for variably sized packets, the average packet size of each flow must be known in advance; making it unsuitable for applications where this is hard to predict. More effective scheduling disciplines, such as deficit round robin and weighted fair queuing were introduced to handle the limitations of WRR. Figure 3 gives an example of weighted round robin queuing; because packets sent are rounded up, each round two packets will be sent from flow 1 and one packet from flow 2.

III. Related Work

The predecessor to `ns-3`, `ns-2` [9], had implemented some scheduling algorithms such fair queuing, stochastic fair queuing, smoothed round robin, deficit round robin, priority queuing, and class based queuing as official modules. `Ns-2` and `ns-3` are essentially different and incompatible environments, `ns-3` is a new simulator written from scratch and is

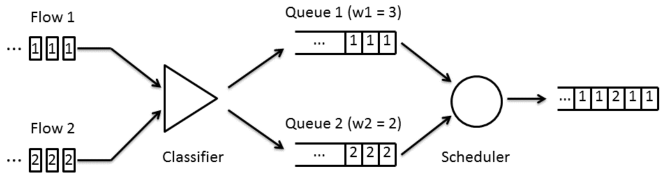


Figure 3. A weighted round robin queue

not an evolution of ns-2. At the time of writing, the latest version, ns-3.32, contains no official differentiated service queuing modules. Several modules have been contributed by others, such as a leaky bucket queue implementation [2] and the previously mentioned DiffServ evaluation module [3].

IV. Design and Implementation

In the DiffServ architecture, there is a distinction between edge nodes, which classifies packets and set the DS fields accordingly, and internal nodes, which queue these packets based on their DS value. In the design framework, we used for all modules, each queue operates independently; we do not utilize the DS field and packets are reclassified at each instance.

WFQ, WRR, and SPQ all inherit from the *Queue* class in ns-3. *Queue* provides a layer of abstraction that hides the scheduling algorithm and allows easy utilization of our classes wherever the *Queue* class or any of its inherited classes exist.

The *Queue* API has three main public members related to functionality: *Enqueue()*, *Dequeue()*, and *Peek()*. In the Point To Point module, *PointToPointNetDevice* passes outgoing packets to *Queue::Enqueue()* when it has finished processing them. *PointToPointNetDevice* calls *Queue::Dequeue()* when the outgoing link is free and begins transmitting the returned packet. Our modules were built specifically with Point To Point in mind, but can be included with any *ns-3* module that utilizes *Queue*.

Classes that inherit from *Queue* must implement the abstract methods *DoEnqueue()*, *DoDequeue()*, and *DoPeek()* which are called by the public methods *Enqueue()*, *Dequeue()*, and *Peek()* respectively.

DoEnqueue() takes a packet as an argument, attempts to queue it, and indicates whether the packet was successfully queued or dropped. *DoDequeue()* takes no arguments, attempts to pop the next scheduled packet, and returns the packet if successful or an error otherwise. *DoPeek()* takes no arguments and returns the next scheduled packet without removing it from the queue.

Our classes follow the same functional design pattern: *DoEnqueue()* calls *Classify()* which determines the correct queue based on user provided parameters. *DoDequeue()* and *DoPeek()* both implement the module-specific scheduling algorithm and return the next the scheduled packet.

To handle user provided criteria for classification, we created an input format modeled after Cisco System's IOS configuration commands. For each of our modules, the classifier sorts incoming packets into separate classes based on these criteria: source IP address, source port number, destination IP address, destination port number, and TCP/IP protocol number (TCP or UDP).

The source and destination address criteria could be either a single host or a range of IP addresses. An optional subnet mask can be provided along with the criteria to distinguish the incoming packets from a particular network. We chose to adopt an inverse mask instead of normal mask (0.0.0.255 instead of 255.255.255.0) for consistency with Cisco IOS.

Each set of user-defined match criteria is stored as an Access Control List (ACL). An ACL consists of a set of entries, where each entry is a combination of the mentioned five-tuple values to uniquely identify a group of packets. After ACLs are introduced to the system, each ACL is linked to a CLASS, which matching packets are associated to. Besides an ACL, a CLASS also has attributes such as weight and queue size. Each instance of CLASS must have at least one associated ACL and each ACL can only relate to one class.

Upon arrival of a new packet, our module attempts to classify the packet into an existing CLASS based on ACLs. If a match is found, the packet is placed into the reserved queue for the corresponding CLASS, however if a match is not found, it will be grouped into the predefined default CLASS. Each reserved queue is a first-in first-out queue with a tail drop policy. An example for configuring the ACL input file is included in the usage section.

A. Strict Priority Queuing

1) Design: SPQ has two internal queues, Q1 and Q2, Q1 is the priority queue and Q2 is the default queue. A single port or IP address can be set by the user and matching traffic is sorted into the priority queue, all other traffic is sorted into the lower priority default queue.

In some SPQ implementations, outgoing regular priority traffic will be preempted in mid-transmission by the arrival of an incoming high priority packet. We chose to only implement prioritization at the time packets are scheduled.

2) Implementation: DoEnqueue() calls the function *Classify()* on the input packet to get a class value. *Classify()* checks if the packet matches any of the priority criterion and indicates priority queue if it does or default queue if it does not. The packet is pushed to the tail if there is room in the queue; otherwise, it is dropped.

DoDequeue() attempts to dequeue a packet from the priority queue. If the priority queue is empty then it will attempt to schedule a packet from the regular queue for transmission.

B. Weighted Fair Queuing

1) Design: Our class based WFQ assigns each packet a class on its arrival. Each class has a virtual queue with which packets are associated. For the actual packet buffering, they are inserted into a sorted queue based on their finish time values. Class (and queue) weight is represented by a floating point value.

A WFQ's scheduler calculates the time each packet finishes service under GPS and serves packets in order of finish time. To keep track of the progression of GPS, WFQ uses a virtual time measure, $V(t)$, as presented in (4). $V(t)$ is a piecewise linear function whose slope changes based on the set of active queues and their weights under GPS. In other words, its slope changes whenever a queue becomes active or inactive.

Therefore, there are mainly two events that impact $V(t)$: first, a packet arrival that is the time an inactive queue becomes active and second, when a queue finishes service and becomes inactive. The WFQ scheduler updates virtual time on each packet arrival [7]. Thus, to compute virtual time, it needs to take into account every time a queue became inactive after the last update. However, in a time interval between two consecutive packet arrivals, every time a queue becomes inactive, virtual time progresses faster. This makes it more likely that other queues become inactive too. Therefore, to track current value of virtual time, an iterative approach is needed to find all the inactive queues, declare them as inactive, and update virtual time accordingly [7]. The iterated deletion algorithm [10] shown in Figure 4 was devised for that purpose.

```
while true do
  $F$ = minimum of $F^\alpha$
  $\delta = t - t_{chk}$
  if $F <= V_{chk} + \delta * \frac{L}{sum}$ then
    declare the queue with $F^\alpha = F$ inactive
    $t_{chk} = t_{chk} + (F - V_{chk}) * \frac{sum}{L}$
    $V_{chk} = F$
    update sum
  else
    $V(t) = V_{chk} + \delta * \frac{L}{sum}$
    $V_{chk} = V(t)$
    $t_{chk} = t$
    exit
  end if
end while
```

Figure 4. The iterated deletion algorithm

Here, α is an active queue, F^α is the largest finish time for any packet that has ever been in queue α, sum is the summation of the weights of actives queues at time t, and L is the link capacity.

We maintain two state variables: t_{chk} and $V_{chk} = V(t_{chk})$. Because there are no packet arrivals in $[t_{chk}, t]$, no queue can become active and therefore sum is strictly non-increasing in this period. As a result a lower bound for $V(t)$ can be found as $V_{chk} + (t - t_{chk}) * \frac{L}{sum}$. If there is a F^α less than this amount, the queue α has become inactive some time before t. We find the time this happened, update t_{chk} and V_{chk} accordingly and repeat this computation until no more queues are found inactive at time t.

After the virtual time is updated, the finish time is calculated for the arrived packet and it is inserted into a priority queue sorted by finish time. To calculate the packet's finish time, first its start time under GPS is calculated, which is equal to the greater of current virtual time and largest finish time of a packet in its queue or last served from the queue. Then, this amount is added to the time it takes GPS to finish the service of the packet. This is equal to packet size divided by weight.

2) Implementation: Similarly to SPQ's implementation, *DoEnqueue()* calls *Classify()* on input packets to get a class value. *Classify()* returns the class index of the first matching criteria, or the default index if there is no match. This class value maps to one of the virtual internal queues, if the queue is not full the packet is accepted, otherwise it is dropped.

Current virtual time is updated as previously described and if the queue was inactive it is made active. The packet start time is calculated by (2) using updated virtual time and queue's last finish time. Then packet finish time is set by *CalculateFinishTime()*. This method uses (3) to return the virtual finish time. The queue's last finish time is then updated to the computed packet finish time. Finally the packet is inserted into a sorted queue based on the finish numbers. A *priority_queue* from C++ container library was used for that purpose.

DoDequeue() pops the packet at the head of priority queue. This packet has the minimum finish time number.

C. Weighted Round Robin

1) Design: WRR has the same number of internal queues, assigned weight representation, and classification logic as WFQ. The weight must be first normalized with respect to packet size. In an environment with variably sized packets, weighted round robin needs to assign a mean packet size s_i for each queue. These parameters are identified by the prior to the simulation in order to correctly normalize the weights. The normalized weight and number of packets sent are calculated by (5) and (6).

2) Implementation: Before the start of the simulation, *CalculatePacketsToBeServed()* determines the number of packets sent from each queue using (6). Similar to SPQ and WFQ, *DoEnqueue()* uses *Classify()* to find the class index of the incoming packets and then puts them in the corresponding queue.

DoDequeue() checks an internal counter to track how many packets to send from the queue receiving service. Each time a packet is sent the counter is decremented. If the counter is equal to zero or the queue is empty *DoDequeue()* marks the next queue in the rotation for service and updates the counter to the value previously determined by *CalculatePacketsToBeServed()*.

V. Validation

To validate our WFQ and WRR implementations we ran a series of experiments against each module. For each experiment, we chose a scenario with predictable outcomes for a given set of parameters based on analysis of the scheduling algorithm. Then we ran simulations of the scenario using the module and compared the recorded results with our analytic model.

All our experiments used the six node topology shown in Figure 5. Each sender and receiver pair sends a traffic flow across a single shared link between the two middle boxes. We installed our modules on the outgoing *NetDevice* across the shared link. In order to observe the characteristics of WFQ and WRR, both senders send 1 GB of data in all simulations and the experiments ends after the last packet in either of the flows is received.

The following parameters are constant across our simulations. The senders use the `ns-3` bulk sending application to

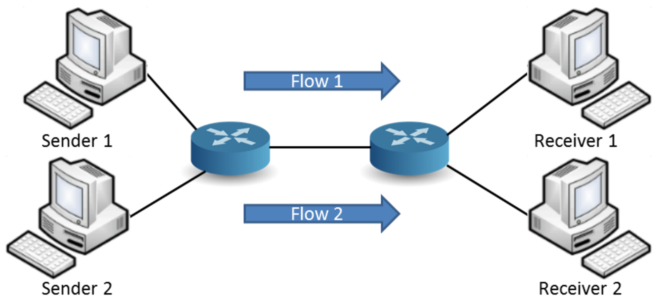


Figure 5. Simulated network used in validation experiments

send 1000 byte packets with no idling, all links have 5ms delay, queue sizes were chosen to be practically unbounded to avoid packet loss.

Because both the weighted fair queuing and weighted round robin are approximations of GPS, and GPS allocates bandwidth based on exact weights, we expect ratio of both traffic flow's throughput to be close to the ratio of weights.

The two traffic flows are assigned weights w_1 and w_2 where $w_2 = 1 - w_2$ for all simulations. Both traffic flows transmit packets at T Mbps from the senders and the shared link has a throughput capacity of $0.5T$, creating a bottleneck. For each module we ran sets of four simulations where $w_2 = 1, \frac{1}{2}, \frac{1}{7}, \frac{1}{10}$ and T is fixed, we repeated this four times with different data rates $T = 0.5, 1, 10, 50$. In each simulation, the receivers measure the average throughput of both flows, R_1 and R_2 over 1ms intervals and the we record the ratio. All simulations stopped after the first traffic flow has finished transmitting.

The results in Figures 6, 7, 8, and 9 show the ratio of throughput at the receivers remains close to the ratio of weights. As we increase data rate across the network, the measured ratio converges to the theoretical one.

For each flow in a correctly implemented WFQ system, the number of bytes served should not lag behind an ideal GPS system by more than the maximum packet length. In low data rates such as 0.5Mbps even one packet can make a noticeable difference. For instance, in a GPS system when the ratio of weights is 10, the first flow sends 10 packets and the second flow sends 100 packets over the same time interval. However, in the corresponding WFQ system if the first flow sends 9 packets, then the perceived ratio will be 11.11 instead of 10.

Because WRR is optimal when using uniform packet sizes, a small number of flows, and long connections we observe that WRR performs as well as WFQ in approximating GPS. As expected, we can see in Figures 10, 11, 12, and 13 that the measured ratio of throughput converges to the ratio of weights as data rate increases.

VI. Usage

CLASSs and their ACLs must be introduced to the simulation before it can begin. Besides working with objects directly in their application, users can provide this information through an XML or text file. In the XML file, using <class_list>and <acl_list>, a hierarchical structure is specifically designed in which users can enter a list of their CLASSs and ACLs separately. The CLASSs and their corresponding ACLs are

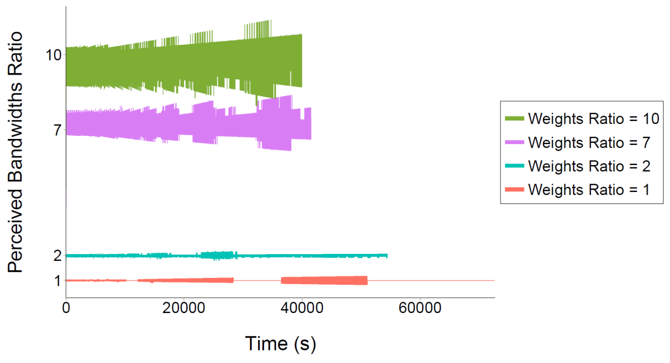


Figure 6. WFQ validation: T = 0.5 Mbps

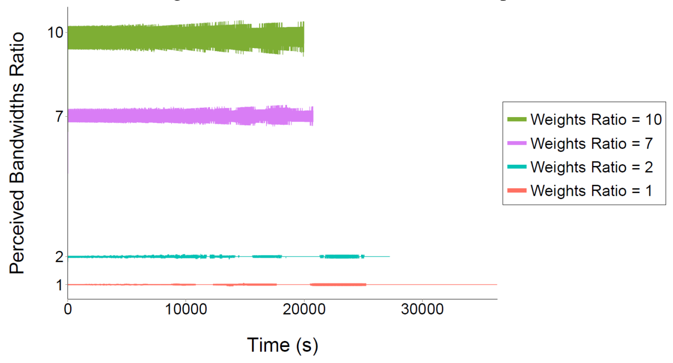


Figure 7. WFQ validation: T = 1 Mbps

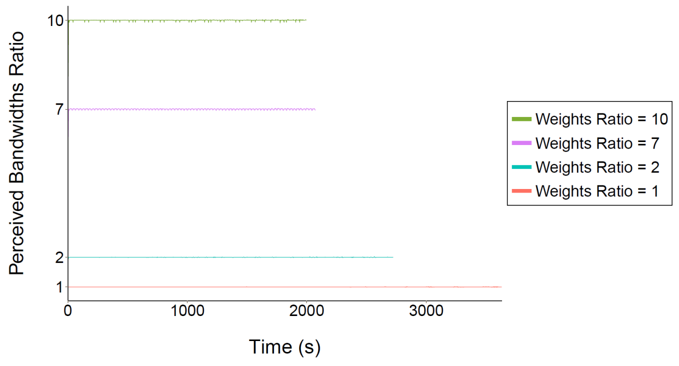


Figure 8. WFQ validation: T = 10 Mbps

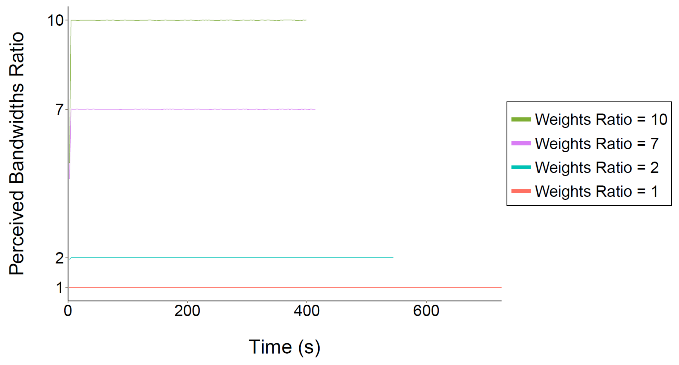


Figure 9. WFQ validation: T = 50 Mbps

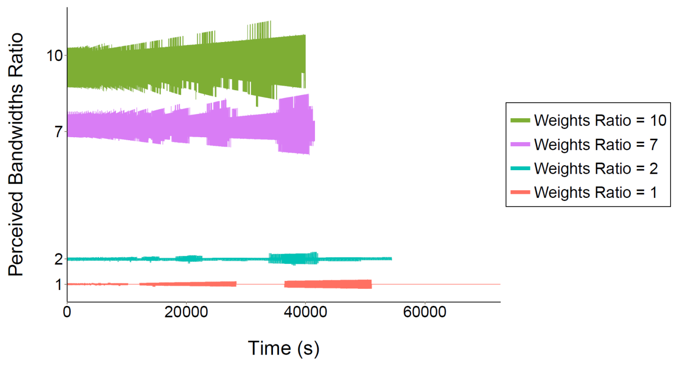


Figure 10. WRR validation: T = 0.5 Mbps

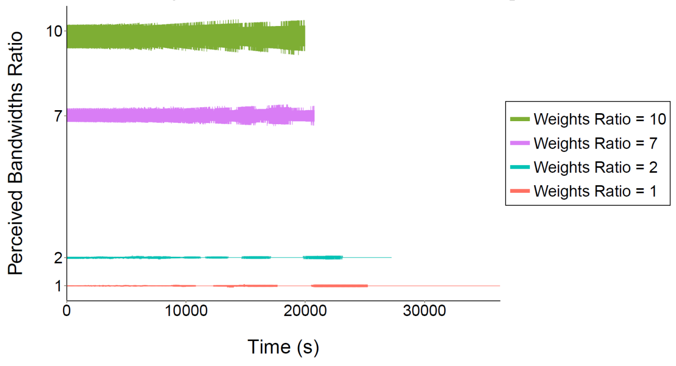


Figure 11. WRR validation: T = 1 Mbps

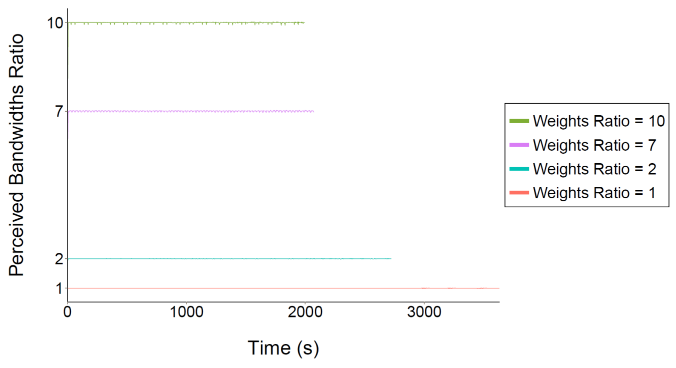


Figure 12. WRR validation: T = 10 Mbps

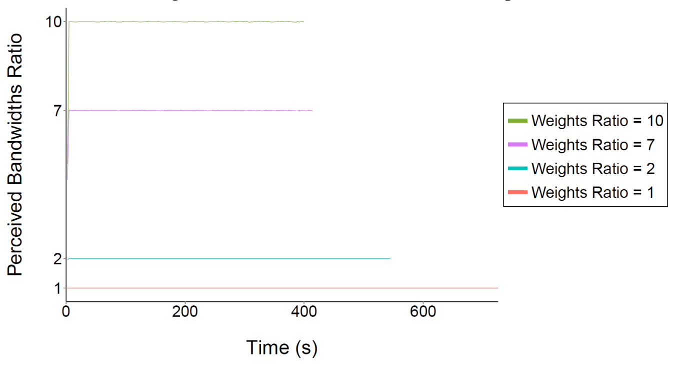


Figure 13. WRR validation: T = 50 Mbps

then linked together using <acl_id>attribute of the class. We have provided an example similar to what we used in our validation scenario:

Alternatively, users can provide the data through a text file. This file consists of a set of lines where each line is a command designated to introduce an ACL or CLASS to the system. The class and access-list commands are used to define a CLASS and an ACL respectively and the class-map command is used to link the two. These commands are simplified versions of Cisco IOS commands and should be familiar to users who have worked with Cisco products.

VII. Conclusion and Future Work

In order to add new functionality to `ns-3`, we have designed and implemented modules for strict priority queuing, weighted fair queuing, and weighted round robin. We have described how these modules correctly implement their respective algorithms within the `ns-3` framework and left the reader with means to utilize them for further experimentation. The ease of configuration and use of our modules should make them attractive tools for further research and we look forward to seeing how others take advantage of our work.

There is a large amount of overlapping functionality between the three queues, particularly WFQ and WRR. All three modules perform basic classification and scheduling at the same points and some of this functionality could be combined into a shared base class for different types differentiated service queues. A stateful classifier and scheduler could be implemented in a child class or associated with the base class as part of a framework for creating these queues.

```
<acl_list><acl id=aclFirstClass><entry>
      <source_address >10.1.1.0</source_address>
      <source_address_mask >0.0.0.255</source_address_mask>
      <source_port_number >23</source_port_number>
      <destination_address >172.16.1.0</destination_address>
      <destination_address_mask >0.0.0.255</destination_address_mask>
      <destination_port_number >23</destination_port_number>
      <protocol>TCP</protocol>
    </entry></acl></acl_list>
```

```
<class_list>
  <class id="class1" acl_id="aclFirstClass">
    <queue_size >256</queue_size>
    <weight >0.875</weight>
  </class>
</class_list>
```

```
access-list access-list-id [protocol] [source_address] [source_address_mask]
    [operator [source_port]] [destination_address] [destination_address_mask]
    [operator [destination_port]]
access-list aclFirstClass TCP 10.1.1.0 0.0.0.255 eq 23 172.16.1.0 0.0.0.255 eq 23
```

```
class [class_id] bandwidth percent [weight] queue-limit [queue_size]
class class1 bandwidth percent 0.875 queue-limit 256
```

```
class-map [class_id] match access-group [acl_id]
class-map class1  match access-group aclFirstClass
```

Figure 14. `class_list`, `acl_list`, `access-list`, `class`, and `class-map`

References

[1] "The ns-3 Network Simulator," Project Homepage. [Online]. Available: http://www.nsnam.org [Retrieved: September, 2015]

[2] P. Baltzis, C. Bouras, K. Stamos, and G. Zaoudis, "Implementation of a leaky bucket module for simulations in ns-3." tech. rep., Workshop on

ICT - Contemporary Communication and Information Technology, Split - Dubrovnik, 2011.

[3] S. Ramroop, “Performance evaluation of diffserv networks using the ns-3 simulator,” tech. rep., University of the West Indies Department of Electrical and Computer Engineering, 2011.

[4] Y. Qian, Z. Lu, and Q. Dou, “Qos scheduling for nocs: Strict priority queueing versus weighted round robin,” tech. rep., 28th International Conference on Computer Design, 2010.

[5] A. K. Parekh and R. G. Gallager, “A generalized processor sharing approach to flow control in integrated services networks: the single node case,” IEEE/ACM Transactions on Networking, vol. 1, no. 3, 1993, pp. 344-357.

[6] A. Demers, S. Keshav, and S. Shenker, “Analysis and simulation of a fair queueing algorithm,” ACM SIGCOMM, vol. 19, no. 4, 1989, pp. 3-14.

[7] S. Keshav, An Engineering Approach to Computer Networking. Addison Wesley, 1998.

[8] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, “Weighted round-robin cell multiplexing in a general-purpose atm switch chip,” IEEE Journal on Selected Areas in Communications, vol. 9, no. 8, 1991.

[9] “The ns-2 Network Simulator,” Project Homepage. [Online]. Available: http://www.isi.edu/nsnam/ns/ [Retrieved: September, 2015]

[10] S. Keshav, “On the efficient implementation of fair queueing,” Journal of Internetworking: Research and Experience, vol. 2, no. 3, 1991.