Simulating Strict Priority Queueing, Weighted Round Robin, and Weighted Fair Queueing with NS-3

Robert Chang and Vahab Pournaghshband

Advanced Network and Security Research Laboratory
Computer Science Department
California State University, Northridge
Northridge, California, USA
bobbychang@google.com
vahab@csun.edu

***Abstract*—Strict priority queueing, weighted fair queueing, and weighted round robin are amongst the most popular differentiated service queueing disciplines widely used in practice to ensure quality of service for specific types of traffic. In this paper, we present the design and implementation of these three methods in Network Simulator 3 (ns-3). ns-3 is a discrete event network simulator designed to simulate the behavior of computer networks and internet systems. Utilizing our implementations will provide users with the opportunity to research new solutions to existing problems that were previously not available to solve with the existing tools. We believe the ease of configuration and use of our modules will make them attractive tools for further research. By comparing the behavior of our modules with expected outcomes derived from the theoretical behavior of each queueing algorithm, we were able to verify the correctness of our implementation in an extensive set of experiments. These implementations can be used by the research community to investigate new properties and applications of differentiated service queueing.**

Keywords–ns-3; network simulator; differentiated service; strict priority queueing; weighted fair queueing; weighted round robin; simulation

I. Introduction

In this paper, we present three new modules for three scheduling strategies: strict priority queueing (SPQ), weighted fair queueing (WFQ), and weighted round robin (WRR). These queueing methods offer differentiated service to network traffic flows, optimizing performance based on administrative configurations.

A. *Network Simulator 3*

The network simulator 3 (`ns-3`) [2] is a popular and valuable research tool, which can be used to simulate systems and evaluate network protocols. `ns-3` is a discrete-event open source simulator. It is completely different from its predecessor `ns-2`. `ns-2` was written in 1995 under the constraints of limited computing power at the time, for example it utilized scripting languages to avoid costly C++ recompilation. `ns-3` is optimized to run on modern computers and aims to be easier to use and more ready for extension. Its core is written entirely in C++. `ns-3` has sophisticated simulation features, such as extensive parameterization system and configurable embedded tracing system with standard outputs to text logs or pcap format. `ns-3` has an object oriented design, which facilitates rapid coding and extension. It also includes automatic memory management and object aggregation/query for new behaviors and state, e.g., adding mobility models to nodes [3].

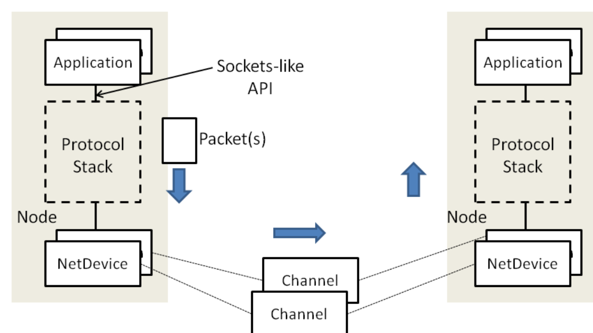


Figure 1. ns-3's simulation network architecture [3]

`ns-3` is aligned with real systems. It has BSD lookalike, event based sockets API and models true IP stack with potentially multiple devices and IP addresses on every node. `ns-3`'s simulation network architecture is similar to IP architecture stack as depicted in Figure 1.

`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 [4] and to add and evaluate a DiffServ framework implementation [5].

B. *Differentiated Services*

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 queueing disciplines, such as those described in this paper,

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

are responsible for executing the flow controls required by DiffServ networks.

DiffServ refers to the differentiated services (DS) field in IP headers. Routers utilize this header to determine which queue to assign each packet in a differentiated service architecture. In addition to this field, there is the older Type of Service (ToS) field in IP headers, which the DS field has largely replaced, and the Class of Service (CoS) field in Ethernet headers. Many enterprise routers utilize these fields to implement the differentiated service methods described in this paper. Routers and switches produced by Cisco and ZyXEL implement SPQ, WRR, and WFQ. Routers and switches produced by Allied Telesis, Alcatel-Lucent, Brocade, Billion Electric, Dell, D-Link, Extreme Networks, Ericsson, Huawei, Juniper Networks, Linksys, Netgear, Telco Systems, Xirrus, and ZTE implement SPQ and WRR. Routers and switches produced by Avaya, Cerio, Hewlett-Packard, RAD, implement SPQ and WFQ. Routers and switches produced by TP-Link implement SPQ only.

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 queueing. 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.

II. BACKGROUND

Each of the modules implemented in this paper is based on well understood queueing algorithm. In this section, we provide an overview on the behavoir of these algorithms.

A. Strict Priority Queueing

Strict priority queueing (SPQ) [6] 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 [6]. Figure 2 gives an example of SPQ; packets from flow 2 cannot be sent until the priority queue is completely emptied of packets from flow 1.

B. Weighted Fair Queueing

Weighted fair queueing (WFQ) [7] 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

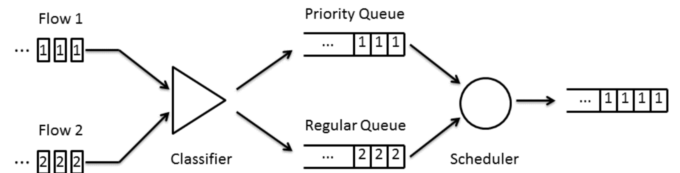


Figure 2. A strict priority queue

bandwidth to each class based on the idealized Generalized Processor Sharing (GPS) model [8].

In the GPS algorithm, the classifier classifies packets from each flow into different logical queues. GPS serves non-empty queues in turn and skips the empty queues. It sends an indefinitely small amount of data from each queue, so that in any finite time interval it visits all the logical queues at least once. This property, in fact, makes GPS an ideal algorithm. Note that if there are weights associated with each queue, then the queues receive service according to their associated weights. When a queue is empty, GPS skips it to serve the next non-empty queue. Thus, whenever some queues are empty, backlogged flows will receive additional service in proportion to their weights. This results in GPS achieving an exact max-min weighted fair bandwidth allocation. While GPS introduces the ideal fairness, it suffers from implementation practicality issues. Because of this issue, numerous approximations of GPS have been proposed that are not ideal but can be implemented in practice. Amongst these proposed GPS approximations are several DiffServ networks such as WFQ and WRR.

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.

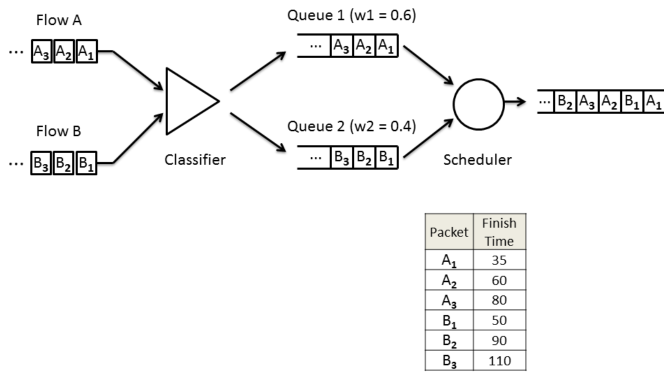
WFQ approximates GPS by calculating the order in which the last bit of each packet would be sent by a GPS scheduler and dequeues packets in this order [9]. The order of the last bits is determined by calculating the virtual finish time of each packet. WFQ assigns each packet a start time and a finish 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).

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

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

$F_i^0 = 0$, 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 $V(t)$

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org



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

Figure 3. A weighted fair queue

is the virtual time at real time t to denote the current round of services in GPS and is defined in (4).

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

$V(0) = 0$, 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 3 gives an example of WFQ; packets are sent in the order determined by their virtual finish times.

C. Weighted Round Robin

Weighted round robin (WRR) queueing 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 variably sized 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) [10].

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

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 WFQ were introduced to handle the limitations of WRR. Figure 4 gives an example of WRR queueing; 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` [11] had implemented some scheduling algorithms such fair queueing, stochastic fair queueing, smoothed round robin, deficit round robin, priority queueing, and class based queueing 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 not an evolution of `ns-2`. At the time of writing, the latest version, `ns-3.23`, contains no official differentiated service queueing modules. Several modules have been contributed by others, such as the previously mentioned leaky bucket queue implementation [4] and DiffServ evaluation module [5].

This paper describes the sames modules with an expanded suite of validation experiments as presented in Chang et al. [1]. In this paper, we present our results from additional experiments performed to further validate the SPQ module and provided a more complete coverage of our WFQ and WRR validation experiments.

Previously, Pournaghshband [12] introduced a new end-to-end detection technique to detect network discriminators (such as the differentiated queueing managements implemented in this paper) and further used the SPQ module presented in this paper to validate their findings. Furthermore, Rahimi et al. [13] introduced a new approach to improve the TCP performance of low priority traffic in SPQ and used the introduced SPQ module in this paper to simulate the approach.

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

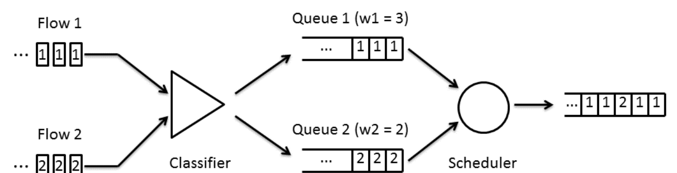


Figure 4. A weighted round robin queue

scheduling algorithm and allows easy utilization of our classes wherever the *Queue* class exists.

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 propose an input format modeled after Cisco System's IOS configuration commands. For each of the disciplines, the classifier sorts incoming packets into separate classes or queues 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. It is 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.

WFQ and WRR use CLASSs for classification purposes. A CLASS has attributes such as weight and queue size. Each instance of CLASS must have an associated ACL and each ACL can only relate to one CLASS.

Upon arrival of a new packet, the classifier 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. Two methods for configuring ACLs and CLASSs via input files are included in the usage section and they are accompanied by actual examples.

As for SPQ, we also suggest an input configuration model. We have not implemented it, however, we briefly explain the idea behind it here. In this model, PRIORITY-LISTs are defined and ACLs are linked to them. A PRIORITY-LIST contains the definitions for a set of priority queues. It specifies which queue a packet will be placed in and, optionally, the maximum length of the different queues. Here, a PRIORITY-LIST has two queues, a high priority queue and a low priority queue. Similar to CLASSs, the same procedure happens when a packet arrives. The classifier tries to put the packet into one of the priority queues based on existing ACLs. If a match is not found, the packet will be placed in the low priority queue. Each reserved queue is a first-in first-out queue with a tail drop policy.

A. Strict Priority Queueing

1) Design: SPQ has two internal queues, which we will refer to as Q1 and Q2, Q1 is the priority queue and Q2 is the default queue. Priority packets are distinguished by either a single port or IP address. Traffic matching this priority citerion are 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. If a priority packet arrives while a regular packet is in transmission, our module will finish sending the packet before scheduling the priority packet.

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 Queueing

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

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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 [9]. 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 [9]. The iterated deletion algorithm [14] shown in Figure 5 was devised for that purpose.

```
while true do
  F = minimum of F^α
  δ = t − t_chk
  if F <= V_chk + δ * L/sum then
    declare the queue with F^α = F inactive
    t_chk = t_chk + (F − V_chk) * sum/L
    V_chk = F
    update sum
  else
    V(t) = V_chk + δ * L/sum
    V_chk = V(t)
    t_chk = t
    exit
  end if
end while
```

Figure 5. 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, WRR 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()*.

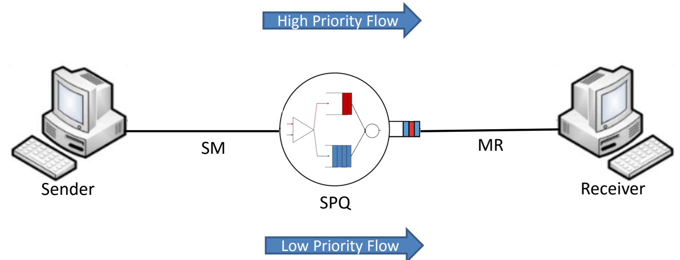


Figure 6. Simulated network used in validation experiments for SPQ

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Algorithm 1 Detecting Strict Priority Queueing

detectStrictPriorityQueueing($n_p = 1000, n_I = 1000, c_p = \{50Mbps, 100Mbps\}$)

1: # Pre-Probing Phase
2: $(\mathcal{P}_{L_j})_{j=1}^{M} \leftarrow$ constructLProbeSequence(n_l, n_p, c_p)
3: $(\mathcal{P}_{H_j})_{j=1}^{M} \leftarrow$ constructHProbeSequence(n_l, n_p, c_p)
4:
5: # Probing Phase
6: send$(\mathcal{P}_{L_j})_{j=1}^{M}$
7: send$(\mathcal{P}_{H_j})_{j=1}^{M}$
8:
9: # Post-Probing Phase
10: $\hat{p}_L^i \leftarrow$ estimateLossRatio($n_l, n_p, ((\mathcal{P}_{L_j})_{j=1}^{M})'$)
11: $\hat{p}_H^i \leftarrow$ estimateLossRatio($n_l, n_p, ((\mathcal{P}_{H_j})_{j=1}^{M})'$)
12:
13: return $(\hat{p}_L^i, \hat{p}_H^i)$

Algorithm 2 Construct Low Priority Probe Sequence

constructLProbeSequnce(n_I, n_p, c_p)

1: $(\mathcal{P}_{L_j})_{j=1}^{M} \leftarrow \emptyset$
2: **for** $i = 1$ **to** n_I **do**
3: $\quad (\mathcal{P}_{L_j})_{j=1}^{M} \leftarrow (\mathcal{P}_{L_j})_{j=1}^{M} \mid\mid (P_H)$
4: **end for**
5: **for** $i = 1$ **to** $(M-1)$ **do**
6: $\quad (\mathcal{P}_{L_j})_{j=1}^{M} \leftarrow (\mathcal{P}_{L_j})_{j=1}^{M} \mid\mid (P_{L_i})$
7: $\quad$ **for** $k = 1$ **to** N' **do**
8: $\quad\quad (\mathcal{P}_{L_j})_{j=1}^{M} \leftarrow (\mathcal{P}_{L_j})_{j=1}^{M} \mid\mid (P_H)$
9: $\quad$ **end for**
10: **end for**
11: $(\mathcal{P}_{L_j})_{j=1}^{M} \leftarrow (\mathcal{P}_{L_j})_{j=1}^{M} \mid\mid (P_{L_M})$
12:
13: return $(\mathcal{P}_{L_j})_{j=1}^{M}$

TABLE I. Parameters used in validating strict priority queueing

Experiment Parameters	Value
Sender to Middlebox link capacity	{50, 100} Mbps
Middlebox Queue Size	20 Packets
Separation Packet Train Length	10 Packets
Initial Packet Train Length	1,000 Packets
Inter-Packet Time	0.1 ns
Packet Size	100 bytes
Middlebox to Reciever Link Capacity	{ 10,...,100 } Mbps

V. VALIDATION

To validate our SPQ, 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 a model that describes the expected behavior of these queueing disciplines.

A. *Strict Priority Queueing*

To verify the correctness of our strict priority queueing implementation we adapted an approach proposed by Pournaghshband [12] to detect the presence of SPQ in a given network topology (Algorithm 1 and 2). In this approach, a sender and a receiver cooperate to detect whether certain traffic is being discriminated using this queueing discipline. In this section, we will validate our implementation for SPQ module in `ns-3` by demonstrating that this approach detects our implemented SPQ accurately.

The original design of SPQ guarantees high priority packets to be scheduled to be transmitted ahead of low priority packets. In the case of high network congestion, this leads to queue saturation, and hence overflow, causing packet losses in any queues. However, due to the inherent nature of packet scheduling in SPQ, in the presence of packets from both high and low priority network flows, the high priority packet loss rate (PLR) is considerably lower compared to that of low priority.

We exploit this starvation issue in presence of high network congestion for low priority network flow and monitor the aggregate loss rate of packets of both high and low packets. We achieve this by constructing a UDP packet probe train consisting of both high and low priority packets. The sender sends this packet train, expecting to saturate both regular and the high priority queues. Algorithm 2 lays out the implementation details of how the packet probe train is created. Figure 8 visualizes how the packets are constructed using Algorithm 2.

We used the network topology shown in Figure 6. Our modified detection application was installed on the sender and the receiver. We set the following fixed parameters for all validation experiments: middlebox queue size to 20 packets, separation packet train length to 2 packets, inter-packet sending time to 0.1 ns, packet size of 100 bytes, and initial packet train length to 1,000 packets. Initial packet train consists of

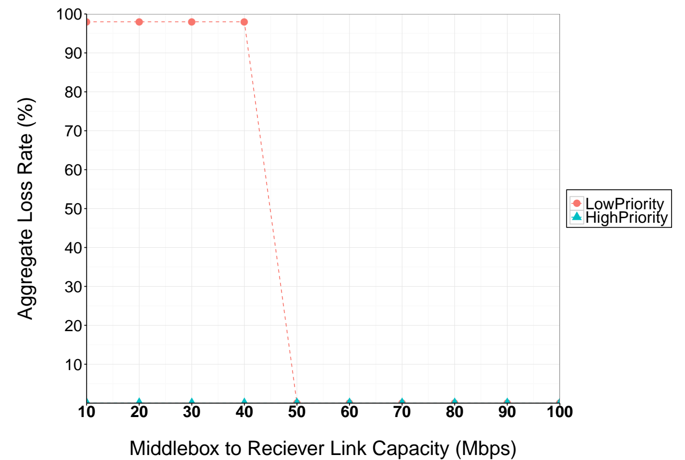


Figure 7. Aggregate loss rate for high priority probes and low priority probes where Sender-to-Middlebox link capacity is 50 Mbps

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

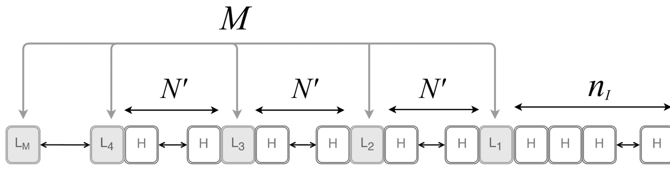


Figure 8. Illustration of packet probe train constructed by ConstructLProbeSeuquence function (N' is the number of packets in the separation packet train. n_I is the initial packet train length. M is the total priority packet probes. (Green) High priority packets. (Red) Low priority packets.)

only high priority packets to saturate the high priority queue in SPQ, which leads to immediate subsequent packets (low or high) to be queued.

We ran two sets of experiments with Sender-to-Middlebox (SM) set to 50Mbps and 100Mbps. In each set of experiments, we used ten different values for Middlebox-to-Receiver (MR) link capacity: 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 Mbps.

All parameters used in experiments validating SPQ are summarized in Table I.

Figures 7 and 9 demonstrate the clear noticeable difference between the perceived aggregate loss rate between high and low priority packets. In the detection algorithm (Algorithms 1 and 2), the packet trains are constructed in a way that satisfies two goals: (1) in High Priority Phase, the separation packet train consisting of low priority packets allows sufficient time for high priority packets to never be backlogged in the high priority queue, resulting in no packet loss for high priority packets. (2) On the other hand, in Low Priority Phase, the separation packet train consisting of high priority packets ensures that the high priority queue is always busy such that the packets in the regular queue never get a chance be served by the scheduler. This should fill up the queue quickly, leading to remaining low priority packets arrived at the queue being dropped.

As a result, as illustrated in Figure 7 and 9, we observed no packet loss for high priority packets in High Priority Phase, and nearly all low priority packets in Low Priority Phase were lost. PLR for low priority packets is not 100% since a few low priority packets (in front of the packet train) were queued in the regular queue, and hence starved but never lost. The exact number of these low priority packets matches the size of the queue (Table I).

We observe the effects of SPQ only when Middlebox-to-Receiver is the bottleneck. This is due to the fact that the service rate at SPQ is lower than the arrival rate. Aggregate loss rate for both high and low priority packets are 0% loss rate when the Middlebox-to-Receiver link capacity is equal to

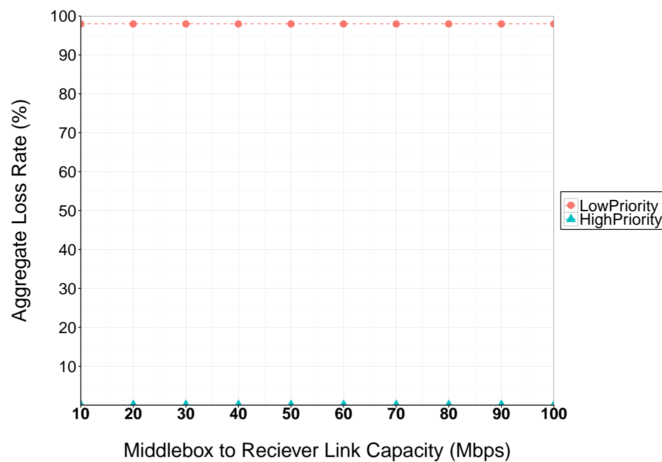


Figure 9. Aggregate loss rate for high priority probes and low priority probes where Sender-to-Middlebox link capacity is 100 Mbps

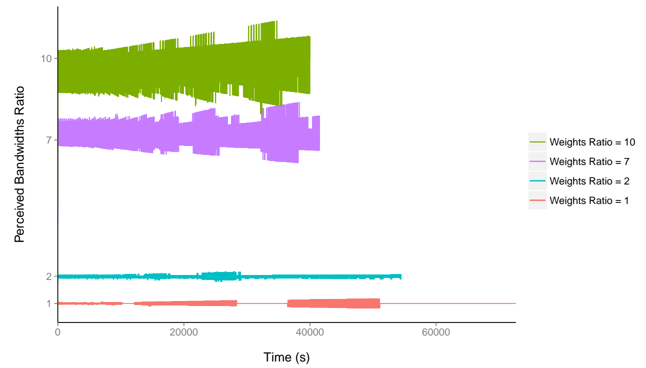


Figure 10. WFQ validation: T = 0.5 Mbps

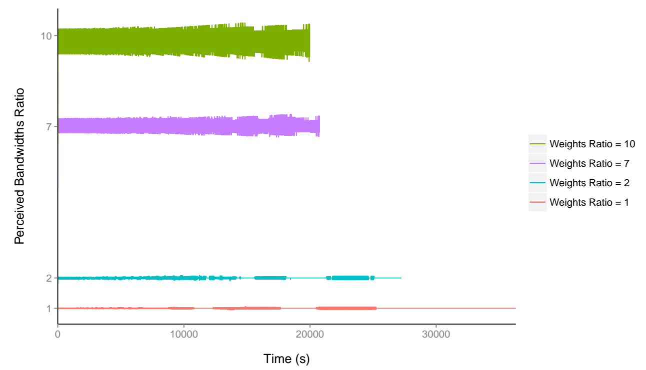


Figure 11. WFQ validation: T = 1 Mbps

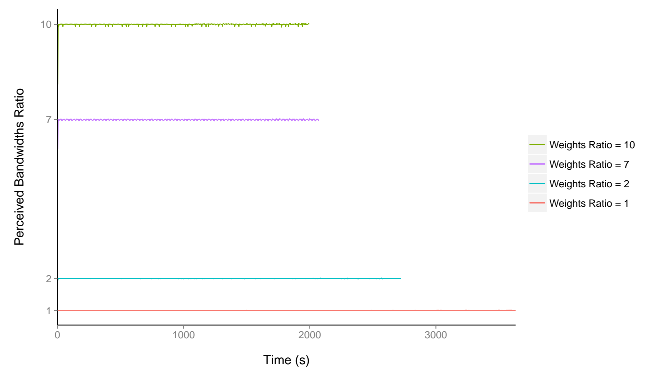


Figure 12. WFQ validation: T = 10 Mbps

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

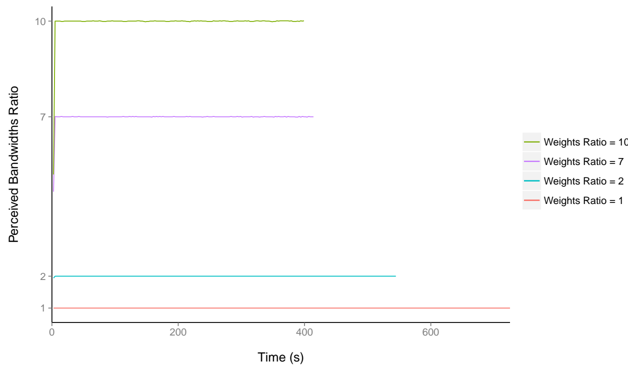


Figure 13. WFQ validation: T = 50 Mbps

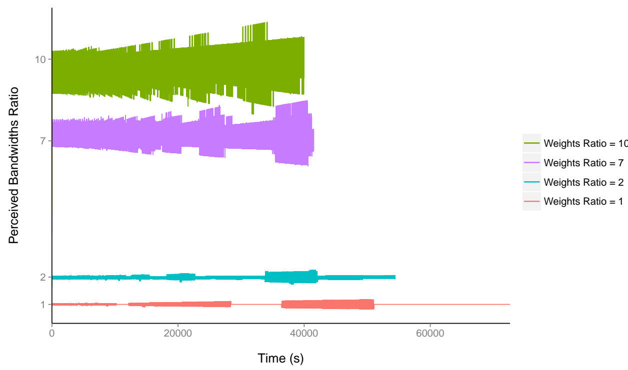


Figure 14. WRR validation: T = 0.5 Mbps

or larger than the Sender-to-Middlebox link. This is when the service rate is the same as or larger than the arrival rate.

In summary, the observed behavior aligns with the expected behavior.

B. Weighted Fair Queueing

Our WFQ experiments used the six node topology shown in Figure 18. The nodes *sender1* and *sender2* each have an instance of `ns3::BulkSendApplication` installed. The nodes *receiver1* and *receiver2* each have an instance of `ns3::PacketSink` installed. *sender1* and *sender2* send a traffic flow across the network to receiver1 and receiver2 respectively. Both traffic flows travel across a single link between the two middle boxes and must share the avaiable bandwidth. We installed a *WeightedFairQueue* module on the outgoing *ns3::NetDevice* across the shared link. In order to observe the characteristics of WFQ, both senders send data at a combined rate above the capacity of the shared link, forcing the *ns3::NetDevice* to utilize the installed queueing module.

The following parameters are constant across all WFQ simulations. The senders use the `ns-3::BulkSendApplication` to send 1000 byte packets with no idling in between and only terminate sending after they have send 1GB of data. All links have 5ms delay, the data rates on all other *ns3::NetDevice* were set high enough to prevent queueing, and the queue sizes on

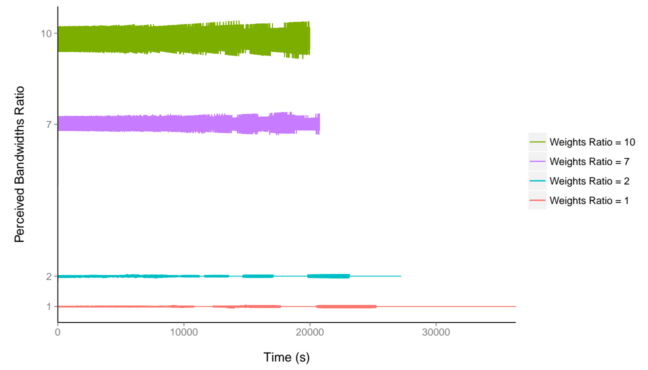


Figure 15. WRR validation: T = 1 Mbps

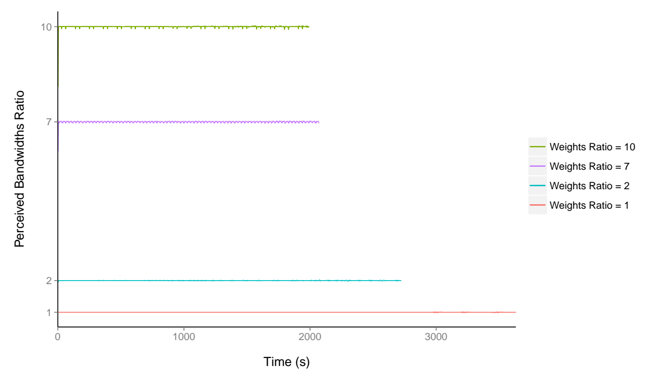


Figure 16. WRR validation: T = 10 Mbps

WeightedFairQueue were set high enough to prevent packet loss. We used *ns3::FlowMonitor* to measure the data rates for each flow.

Because WFQ is 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

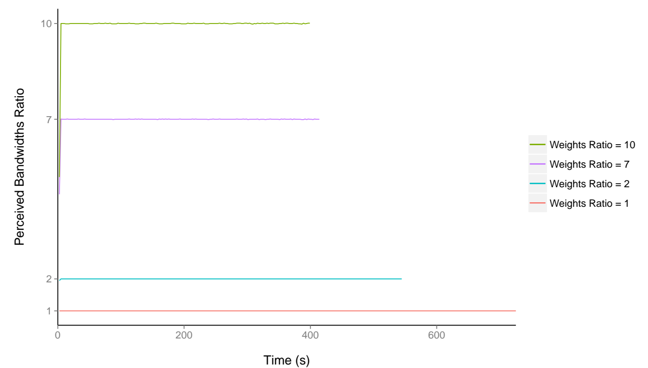


Figure 17. WRR validation: T = 50 Mbps

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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 had finished transmitting to prevent the experiment from recording any data that does not include both senders.

The results in Figures 10, 11, 12, and 13 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.

C. Weighted Round Robin

Our WRR experiments used the same six node topology used by the WFQ experiments, shown in Figure 18. `ns-3::BulkSendApplication` and `ns-3::PacketSink` were installed on the same nodes, and `ns-3::FlowMonitor` was used to collect data. In these experiments, we installed the *WeightedRoundRobin* module on the outgoing `ns3::NetDevice` across the shared link.

Similarly, each `ns-3::BulkSendApplication` was programmed to send 1000 byte packets with no idling until 1 GB of data had been sent. We kept all link delays, data rates, and queue sizes in this set of experiments unchanged from the WFQ experiments.

We collected data from these experiments using the same methods and calcuations described in the previous section on WFQ. Like WFQ, WRR is an approximation of GPS, and we expect ratio of both traffic flow's throughput to be close to the ratio of weights. 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 14, 15, 16, and 17 that the measured ratio of throughput converges to the ratio of weights as data rate increases.

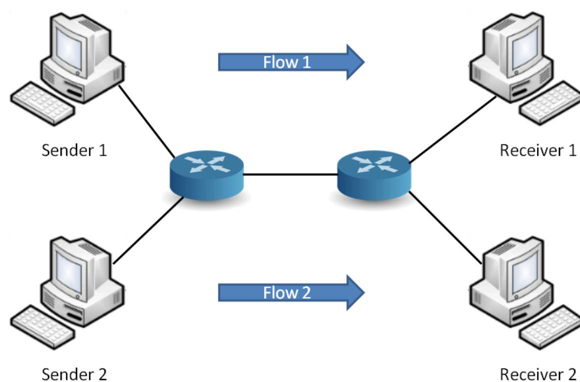


Figure 18. Simulated network used in WFQ and WRR validation experiments

VI. Usage

As discussed in Section IV, CLASSs and their corresponding ACLs must be defined before the simulation can start. In addition to working with objects directly in their applications, users can provide this information through a text or XML file.

A. Text input configuration file

The input configuration data can be provided through a text file. This file consists of a set of lines where each line is a command designated to introduce an ACL or a CLASS to the system. These commands are simplified versions of Cisco IOS commands and should be familiar to users who have worked with Cisco products.

Here, `access-list` command can be used to define a single ACL. Figure 19 shows the syntax of the command and the order of the parameters.

```
access-list access-list-id
    [protocol]
    [source_address]
    [source_address_mask]
    [operator [source_port]]
    [destination_address]
    [destination_address_mask]
    [operator [destination_port]]
```

Figure 19. access-list command syntax

Furthermore, `class` command defines a single CLASS. Weight amount can be determined using the `bandwidth percent` parameter and queue size is set by `queue-limit`. Figure 20 demonstrates the syntax of class command and the order of the parameters.

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

Figure 20. class command syntax

After defining a CLASS, it is expected to be linked to an ACL. This is done through a `class-map` command that matches one CLASS with one ACL. Figure 21 provides the syntax of class-map command.

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

Figure 21. class-map command syntax

We provide a simple example here to configure a sample WFQ system that has two classes with weights' ratio of 7

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

and queue sizes of 256 packets. We define an ACL called `highACL` for the class with higher weight and another ACL called `lowACL` for the class with lower weight. `highACL` includes traffic from 10.1.1.0 network with port number 80 aiming to 172.16.1.0 network with the same port number via TCP protocol. `lowACL`, however, includes traffic from 10.1.1.0 with port number 21 heading towards 172.16.1.0 with the same port number via TCP protocol. Figure 22 demonstrates the example.

```
access-list highACL TCP 10.1.1.0 0.0.0.255 eq 80
    172.16.1.0 0.0.0.255 eq 80
access-list lowACL TCP 10.1.1.0 0.0.0.255 eq 21
    172.16.1.0 0.0.0.255 eq 21
class highCl bandwidth percent 0.875 queue-limit 256
class lowCl bandwidth percent 0.125 queue-limit 256
class-map highCl match access-group highACL
class-map lowCl match access-group lowACL
```

Figure 22. A sample example demonstrating configuration of WFQ using text file commands

B. *XML input configuration file*

Alternatively, users can provide input configuration data via an XML file where they can provide a set of ACLs to the system using `<acl_list>` tag. An <acl_list> tag can include one or multiple ACLs. Each ACL is defined using an `<acl>` tag and can have one or multiple set of rules or entries defined by `<entry>` tags.

Similar to ACLs, CLASSs are introduced using a `<class_list>` tag, which is a set of CLASSs each defined by a `<class>` tag. Each CLASS has `queue_limit` and `weight` properties and is linked to its corresponding ACL using `acl_id` attribute.

Again we provide an example of configuration of WFQ. We use the example that we already used in the previous section and show that how it can be implemented via an XML file. Figure 23 demonstrates the example.

```
<acl_list>
  <acl id=``highACL">
    <entry>
      <source_address >10.1.1.0
       </source_address>
      <source_address_mask >0.0.0.255
       </source_address_mask>
      <source_port_number >80
       </source_port_number>
      <destination_address >172.16.1.0
       </destination_address>
      <destination_address_mask >0.0.0.255
       </destination_address_mask>
      <destination_port_number >80
       </destination_port_number>
      <protocol>TCP</protocol>
    </entry>
  </acl>
  <acl id=``lowACL">
    <entry>
      <source_address >10.1.1.0
       </source_address>
      <source_address_mask >0.0.0.255
       </source_address_mask>
      <source_port_number >21
       </source_port_number>
      <destination_address >172.16.1.0
       </destination_address>
      <destination_address_mask >0.0.0.255
       </destination_address_mask>
      <destination_port_number >21
       </destination_port_number>
      <protocol>TCP</protocol>
    </entry>
  </acl>
</acl_list>
<class_list>
  <class id=``highCl" acl_id=``highACL">
    <queue_limit >256</queue_limit>
    <weight >0.875</weight>
  </class>
  <class id=``lowCl" acl_id=``lowACL">
    <queue_limit >256</queue_limit>
    <weight >0.125</weight>
  </class>
</class_list>
```

Figure 23. A sample example demonstrating configuration of WFQ using XML file

VII. Conclusion and Future Work

In order to add new functionality to `ns-3`, we have designed and implemented modules for SPQ, WFQ, and WRR. We have detailed our implementations of these well known algorithms and presented the reader with the means to understand their operation. We have validated these modules and shared our experiment designs and results to prove their correctness. Utilizing the instruction and examples we have given, readers can write simulations that make use of our modules 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 further oppurtunity to break these modules into abstract components for re-use. There exist obvious shared functionality between the three queues, particularly WFQ and WRR. All three modules perform classification, a new class could be implemented to handle classsification for any differentiated service queue. Scheduling, although different for each queue type, has identical interfaces and could be implemented as a base class for differentiated service queues to inherit from and respective their own scheduling classes. These shared classes could form the base of a framework for creating additional differentiated service queue modules in `ns-3`.

References

[1] R. Chang, M. Rahimi, and V. Pournaghshband, "Differentiated Service Queuing Disciplines in ns-3," In Proc. of the Seventh International Conference on Advances in System Simulation (SIMUL), Barcelona, Spain, November 2015.

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

[3] J. Kopena, "ns3: Quick Intro and MANET WG Implementations," Proceedings Of The Seventy-Second Internet Engineering Task Force, Dublin, Ireland, 2008.

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[4] 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.

[5] 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.

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

[7] A. Parekh and R. 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.

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

[9] S. Keshav, “An Engineering Approach to Computer Networking.” Addison Wesley, 1998.

[10] 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.

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

[12] V. Pournaghshband, “End-to-End Detection of Third-Party Middlebox Interference, Ph.D. Dissertation, University of California, Los Angeles, 2014

[13] M. Rahimi and V. Pournaghshband, “An Improvement Mechanism for Low Priority Traffic TCP Performance in Strict Priority Queueing,” In Proc. of IEEE International Conference on Computer Communications and Informatics (ICCCI), pp. 570, January 2016.

[14] S. Keshav, “On the efficient implementation of fair queueing,” In Journal of Internetworking: Research and Experience, volume 2, number 3, December 1991.

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org