EE 5410 TERM PROJECT

Egress Link QoS Scheduling

Ahmet Emrah Demircan

January 2014

Table of Contents

[Abstract 3](#_Toc378644241)

[Discussion of the Selected Paper, Motivation and Work Done 3](#_Toc378644242)

[Project Software Package 4](#_Toc378644243)

[QoS Scheduler Application 5](#_Toc378644244)

[Kernel Space Module Project 5](#_Toc378644245)

[User Space Application 7](#_Toc378644246)

[Raw Ethernet Tester Application 9](#_Toc378644247)

[Emulation Test Setup 11](#_Toc378644248)

[Test 1 13](#_Toc378644249)

[Test 2 16](#_Toc378644250)

[Test 3 18](#_Toc378644251)

[Test 4 19](#_Toc378644252)

[Results 21](#_Toc378644253)

# Abstract

The purpose of this project is studying and demonstrating the egress link QoS schedulers as discussed in the paper “Providing QoS with the Deficit Table Scheduler”. The major objective of the paper is proposing a table based scheduler satisfying latency requirements more than other table based schedulers while keeping the advantage of being simpler than sorted-priority family schedulers such as WFQ which are also good at fairness and latency issues. This project work consists of a practical work done to discuss some selected parts of the paper. As the paper discusses various aspects and tries to realize the author’s perspective through some simulator work, it has been aimed to criticize the paper without fine-grained considerations while intending to emulate discussed scheduling algorithms and evaluate them taking fairness, latency and complexity requirements into consideration.

# Discussion of the Selected Paper, Motivation and Work Done

QoS scheduling on the egress links is the objective of the paper. The authors, while considering the high speed networks and high performance requirements, are denoting the need for proposing and realizing an egress link QoS scheduler which is fair in bandwidth allocation, good in latency requirements and most importantly low in complexity. Complexity reduction efforts are coming from the fact that scheduling time for a packet should be smaller than the average packet transmission time. Thus, contemporary high speed links should be coupled with efficient scheduling algorithms. This issue is well discussed throughout the paper while explaining different QoS schedulers comparatively.

The QoS schedulers are discussed in many aspects in the paper. A couple of categorizations have been made by giving references from the research area. Among those categorizations, “table based” vs. “sorted priority”, “work-conserving” vs. “non-work-conserving” and “frame based architectured” vs. “sorted priority architectured” schedulers can be named. A discussion based on the frame based or sorted priority architectures have also been made and both advantages and disadvantages of both categories have been stated:

* Sorted priority architrectured schedulers (WFQ, WF2Q, etc.) generally provide good fairness and a low latency while they cannot be considered as efficient in complexity measures.
* Frame based schedulers (including the round robin family) can provide good fairness and a low latency provided that the frames are distributed along an average frame length. These schedulers offer very low complexity features as an advantage on the other hand. Of course there are some modified versions such as deficit round robin scheduler which are providing a good fairness and low latency measure in the average.

However, there is a latency related problem stated by the authors which leads to proposal of a new scheduler. They put forward the dependency of latency measure to the bandwidth assignment policy issue and state that in frame based schedulers (which are actually somewhat table based and slotted); latency provided to the different flows is completely dependent on the bandwidth assigned to each flow. Bandwidth demanding flows can capture the line for long intervals without giving way to low bandwidth assigned flows in burst traffic. Thus, those packets from less bandwidth assigned flows are prone to more latency than the others. This is the main problem addressed in the paper and the authors are proposing a new table based “Deficit Table (DTable) Scheduler” to solve this issue.

DTable scheduler has an algorithm very similar to the deficit round robin scheduler and it works on credit collection logic based on priorities and flow weights. But the main difference is that the frame in DTable is based on an arbitration table considering weights (just like weighted round robin) and the credit collection cycle is distributed over a fine-grained timeline. This means that the flows are collecting credits part by part in one cycle opposed to a bulk (bulk may not be the right word here) collection method we see in the deficit round robin scheduler. This method gives a chance of transmission to low priority flows which have access slots scattered on the arbitration table. Of course all these modifications have been made for variable packet sized flows on the network. Here is a comment from the paper:

*“Note that if we distribute all the entries belonging to the same flow in consecutive way in the arbitration table, the performance of the scheduler is going to be similar to the DRR scheduler. ... Therefore if we want to improve the latency performance provided by this scheduler, we can distribute the table entries as the WF2Q variant of the list-based Weighted Round Robin proposed by Chaskar and Madhow”*

The authors have also put their practical work forward to support their proposal. There are many simulation results and statistical work added to the discussion. The motivation for this project is the need for creation of an emulation environment to test the proposed scheduler with real packets and transmissions on the line. As the paper has been analyzed in a criticizing way, although the simulation results have been supporting the proposed scheduler, it would have been good to implement and test the schedulers on a general purpose processor based environment.

But, there are some drawbacks of working on a non-real time environment. These are high resolution timer needs, real-time thread scheduling issues, network driver access methods and some other operating system related complications. Though, a pretty good infrastructure has been built up and a test setup for evaluating 6 different schedulers has been possible.

The scheduler related part of the project is designed on a Linux OS environment. The software prepared uses Linux network stack handles efficiently to run schedulers. On the other hand, testing software has been prepared on a Windows OS environment to feed the scheduling processes. Various tests have been made with the setup as some of them are also shared in this report.

# Project Software Package

The term project software package consists of 2 applications:

* The QoS Scheduler Application: This Linux based application is the main software item developed for implementing various egress link QoS schedulers, showing real-time statistics about the scheduling and flow behaviors.
* Raw Ethernet Ethernet Tester Application: This Windows application has been developed for generating different ethernet based flows for feeding the QoS Scheduler Application on Linux.

## QoS Scheduler Application

This software item has been developed on Qt Creator IDE to work on Linux environment. It consists of two software modules working in kernel and user space respectively. Thus it consists of two different Qt projects, one for the kernel space (module\_5410.pro) and the other for the user space (user\_5410.pro). Although there exist two projects it has been preferred to be called as one application (Figure 1 shows the user interface of the application) as both cooperate to realize the QoS Scheduler on Linux.

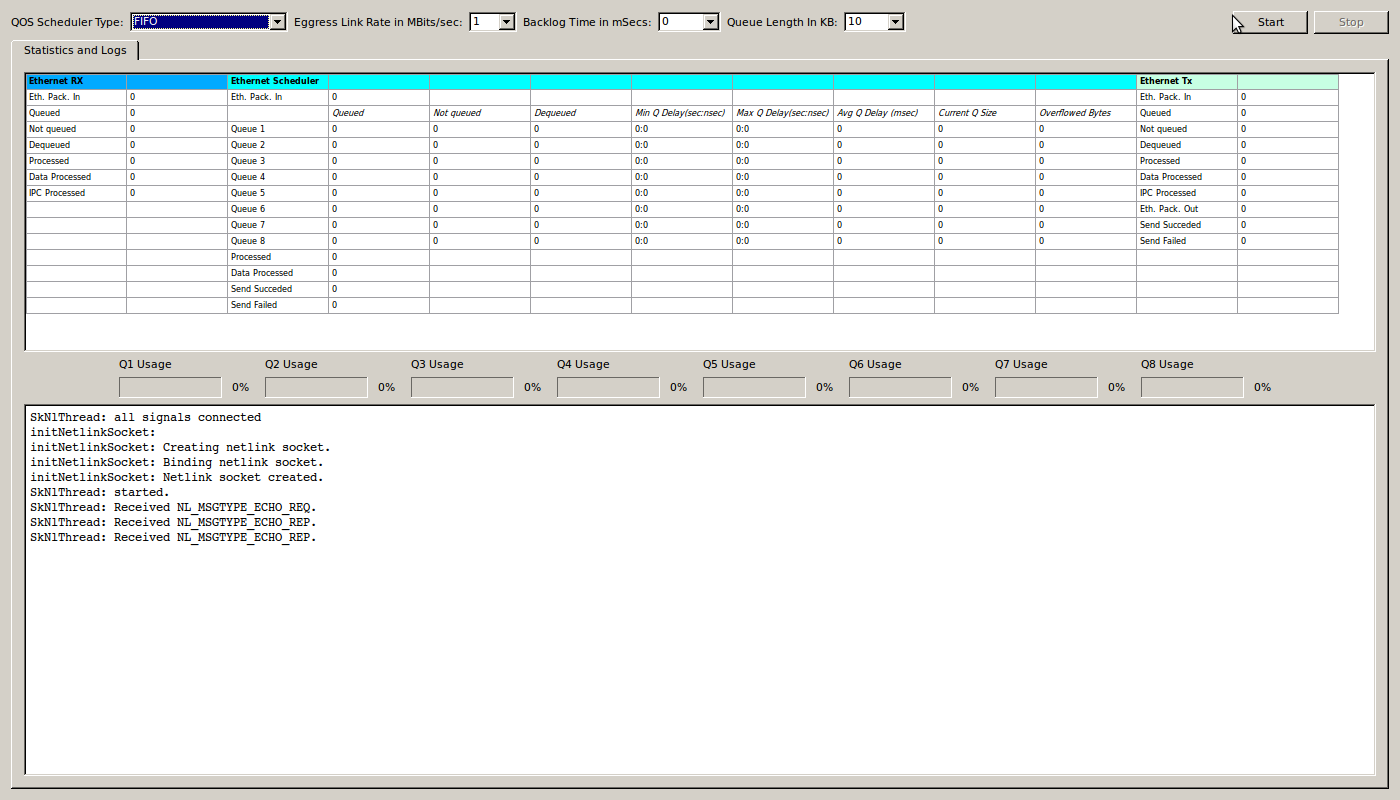


Figure 1: QoS Scheduler Application user interface

### Kernel Space Module Project

The kernel space project module\_5410.pro is for building a kernel module realizing ethernet level packet sniffing, flow classification, flow based QoS scheduling and egress link transmission functionalities.

A flow is defined as the set of ethernet packets with the same ethertype field values. Such a simple identification method has been selected so that the implementation simplicity for accessing ethernet packets on Linux network stack is used. There are 8 different flows defined statically in the software package:

|  |  |
| --- | --- |
| **Ethertype Value** | **Priority** |
| 0x8800 | 1 |
| 0x8801 | 2 |
| 0x8802 | 3 |
| 0x8803 | 4 |
| 0x8804 | 5 |
| 0x8805 | 6 |
| 0x8806 | 7 |
| 0x8807 | 8 |

Table 1: Ethertype based flows and priorities

The priorities assigned to the ethertype values are defined as “highest the number prior the flow”.

The kernel space project consists of 3 functional sub-units working on the ethernet flows as defined below:

* Ethernet Receiver: This sub-unit is responsible for attaching the Linux network stack packet interface through a software interrupt routine. It filters and queues the ethernet packets with predefined ethertypes (see Table 1) and then processes and sends each packet to the ethernet scheduler sub-unit through its kernel thread context. This sub-unit basically steals ethernet packets received from the ingress network interface and passes them to the scheduler.
* Ethernet Scheduler: This sub-unit is the main functional unit in scope of the project. Flows are classified and scheduled by this unit. The scheduling sub-unit keeps ethernet packets in queues which can be sized by user selection. Thus packet overflows may appear or not according to the burstiness of the flows and the queue sizes given. On the other hand, this unit has the capability to downgrade the egress link rate to user selected values. The scheduling is made on flows by 6 algorithms:
  + FIFO: Only one reception queue is kept for scheduling purposes and all packets coming from the ethernet receiver sub-unit are queued in that unique queue. All flows are of same priority here and nothing special is done for distinct ethertype-priority matching. Thus, there is no scheduling and all ethernet packets are sent to the ethernet sender sub-unit one by one in a first in first out manner. The queue length used for this scheduler is 8 times the user selected queue length to be comparable with the other algorithms having special queues for each flow.
  + Round Robin: There are 8 flow queues kept for each ethertype defined in Table 1. All packets coming from the ethernet receiver sub-unit are queued in the corresponding flow queue to be processed later in the scheduler kernel thread. The scheduler kernel thread visits each flow queue one by one and picks and sends ethernet packets to the ethernet sender sub-unit.
  + Weighted Round Robin: This scheduling algorithm takes flow priorities in concern and picks and sends ethernet packets by using a cyclic table built up by using weights and flow identifications. The most prior flow has the biggest ethertype value (0x8807), thus the queue for 0x8807 flow is visited the most. One cycle consists of visits and the most prior flow has 8 visits while the least prior has 1 visit in each cycle. Picked ethernet packets are sent to the ethernet sender sub-unit as usual.
  + Deficit Round Robin: This scheduling algorithm takes varying packet lengths into concern and tries to make a fair bandwidth allocation for the flows. There are 8 queues handled for scheduling. Each cycle consists of one visit per each flow, credit updates and packet pick-ups if credits are enough for the flow. The ethernet packets are forwarded to the ethernet sender sub-unit.
  + Weighted Fair Queuing, WF2Q: This is the only virtual time based scheduling algorithm among schedulers implemented in this project. Each enqueue and dequeue event is used to calculate the scheduler virtual time and start/finish times of each packet if available at the head of classified flows. This is the fairest algorithm with the highest computational complexity.
  + Deficit Table Scheduling: This scheduling algorithm is proposed in the selected paper. It is a Deficit Round Robin like scheduling algorithm which is modified by incrementing and also spanning the flow visits just like the weighted round robin scheduling algorithm. The configuration of this scheduling algorithm is also proposed as a methodology in the paper, but this issue is left untouched in scope of this project.

As discussed before, scheduling algorithms are using the flow queues that can be sized with user inputs. There is also another parameter which delays scheduling startup time by some milliseconds just to create a backlog period for emulating bursty traffic. Actually this functionality would have been altered by a traffic generator with burst capability but it has been preferred to be kept in the scheduling startup logic.

The egress port rate degradation logic slows down the scheduled packet transfer to the ethernet sender sub-unit by using Linux high resolution timer interrupts. Each packet’s length is considered to calculate a transmission time to block the TX feeding for some time as the ethernet sender sub-unit uses asynchronous non-blocking driver calls. So, the scheduler is logically obeying the transmission delays and traffic bursts can easily be created by selecting an egress port transmission rate lower than the traffic rate on the ingress port.

* Ethernet Sender: This sub-unit is the last layer on time-line. It is responsible to keep scheduled ethernet packets in one output queue and send each packet to the output interface one by one through a kernel thread. The transmission request to the Linux network stack are non-blocking but there is overflow protection to speed down the sending process. The unique functionality of this sub-unit includes modification of the ethertype field, i.e. incrementing the ethertype value by 0x1000, and transmission of it. For instance, a scheduled ethernet packet from flow 0 with ethertype 0x8800 will be modified to have an ethertype 0x9800 on the egress link. This is a requirement for distinguishing the outgoing and incoming ethernet packets as seen from the sniffer machine. Wireshark on the sniffer machine will display outgoing flows with [0x8800,0x8807] ethertype range and scheduled flows with [0x9800,0x9807] ethertype range.

Apart from all, the kernel module keeps track of various statistics to be shown to the user. These are packets queued in and out, online queue sizes, overflowed bytes, queuing delays for each flow. These statistics are continuously transmitted to the user space application to be dumped on the screen.

### User Space Application

The user space project user\_5410.pro is for building a user space GUI (**G**raphical **U**ser **I**nterface) for controlling the kernel module and informational monitoring such as dumping kernel module sourced statistics and logs. The user space application “user\_5410” (see Figure 1) installs the kernel module object “module\_5410.ko” at startup and after necessary initializations passes the control to the user.

The user can, then, select from available QoS Scheduling Algorithms and start it for testing. User selectable controlling parameters are as follows:

* The scheduling algorithm: There are 6 scheduling algorithms implemented on the kernel module. User can select from FIFO, Round Robin, Weighted Round Robin, Deficit Round Robin, WF2Q, Deficit Table schedulers (Figure 2).

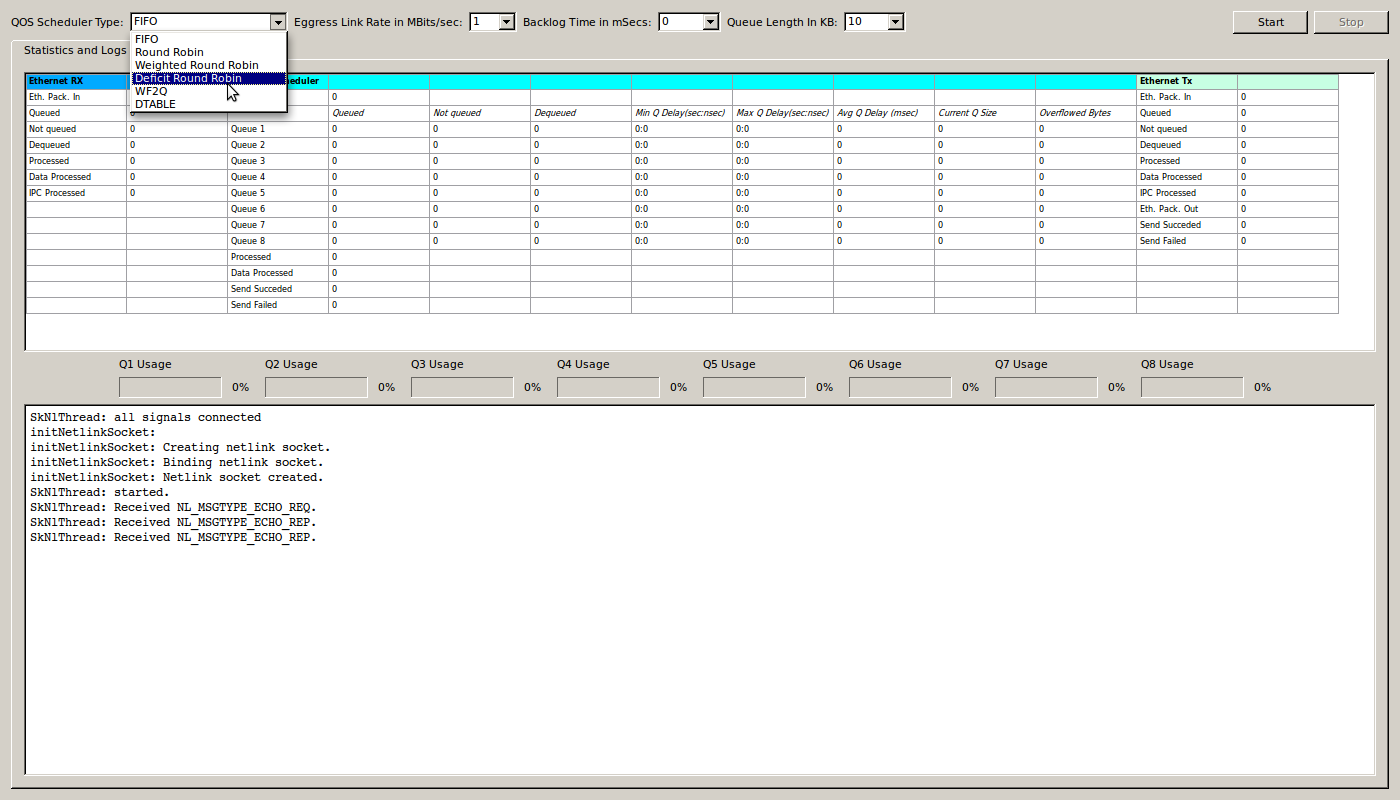


Figure 2: Selecting scheduler

* Egress Link Rate: This parameter is used to limit the scheduler packet transmission by rate degradation. The physical interface link speed remains the same but; the selected rate value is used to slow down the transmission in software.

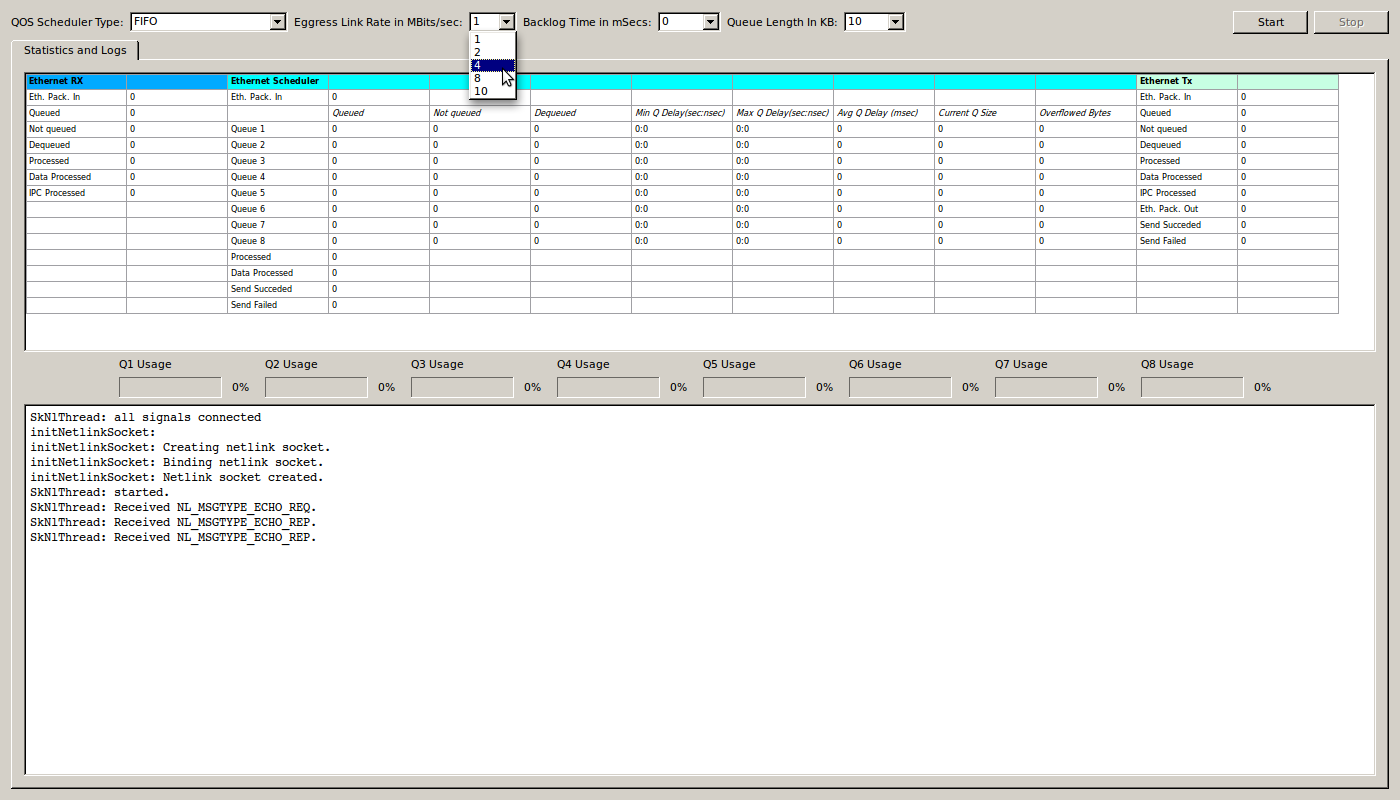


Figure 3: Selecting egress link rate

* Backlog Time: This parameter is used for delaying the scheduler kernel thread by milliseconds granularity at the startup. Delaying the ethernet scheduler sub-unit while the ethernet receiver sub-unit is processing fills in the queuing buffers for the backlog time period and emulates a bursty traffic effect as if all flows have ramped up in transmission.

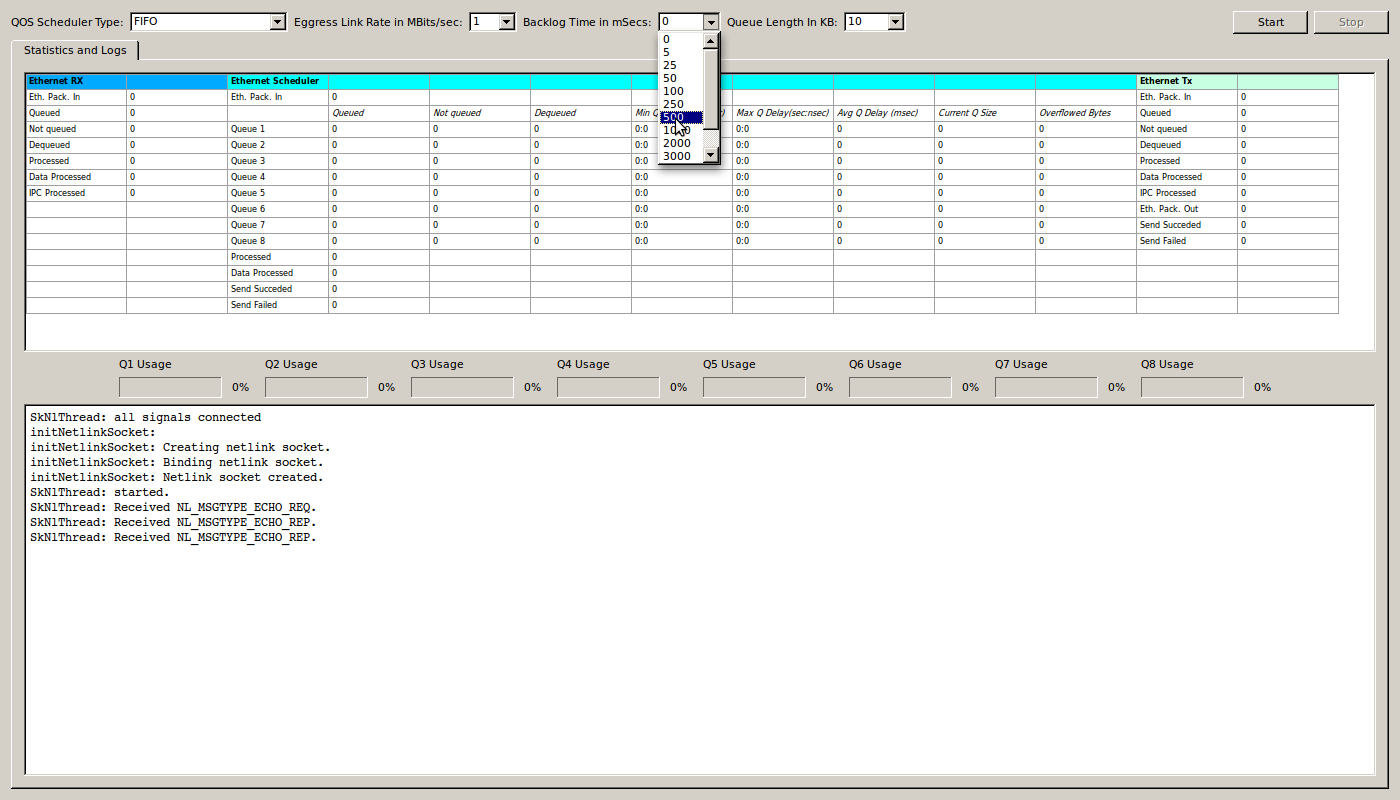


Figure 4: Selecting backlog time

* Queue Size: This parameter is used for resizing the flow assigned queues to emulate the case that packets for unscheduled flows are dropped in bursty traffic.

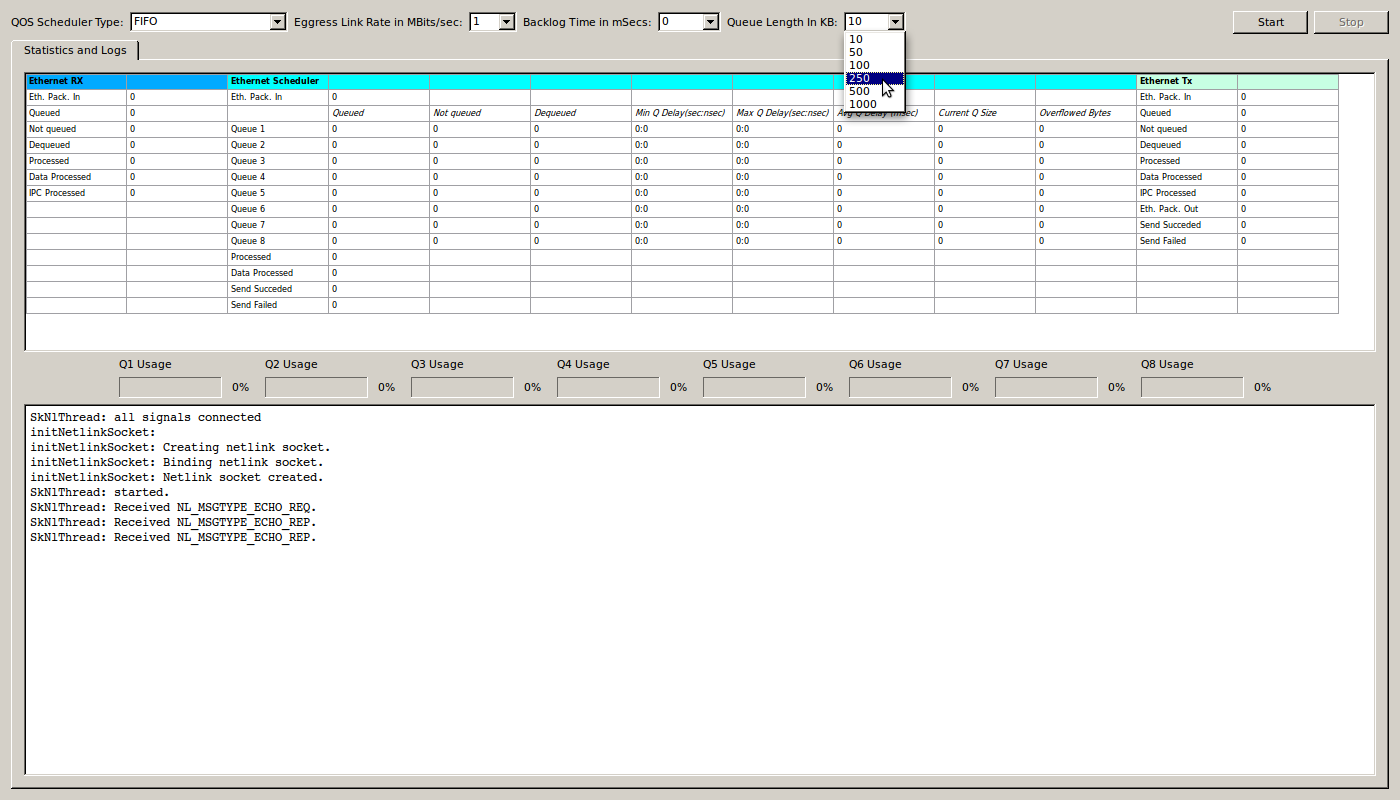


Figure 5: Selecting Queue Length

The user space application talks with the kernel module through a netlink socket and exchanges commands through that interface. Passing selected scheduler, backlog time, queue size, scheduling start and stop request are all directed from the user application to the kernel module. On the other hand, kernel module passes all statistical information and logs to the user application periodically.

## Raw Ethernet Tester Application

This software item has been developed on Borland C++ Builder development environment to accompany the Linux QoS Scheduling Application as an input for the tests. This GUI application (Figure 6) realizes the creation of different user defined ethernet flows and the transmission of ethernet packets belonging to them. This is not a traffic generator application with much functionality but is quite enough for the QoS scheduling test input.

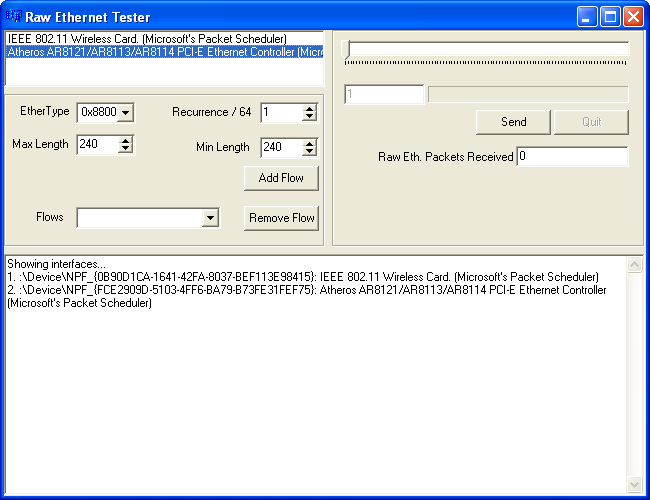


Figure 6: Raw Ethernet Tester application

The packet generation is a user defined number of repetitions defined as cycles. Each cycle consists of at most 64 slots to be pre-allocated by user defined flow recurrences as a transmission scenario. Although the cycle behavior is strictly constant, each cycle differs by the transmission of packets with varying payloads that are instantaneously and randomly selected for corresponding flows. The logic behind flow generation is very simple:

* First of all, the flows are classified by the ethertype field in the ethernet header. There are 8 different and constant ethertype values which can be used for labeling flows (Figure 7).

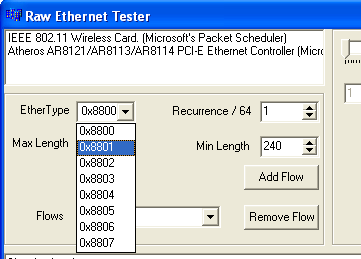


Figure 7: Selection of ethertype for a flow

* Secondly, each flow can generate ethernet packets with a random payload length in between a minimum and a maximum value (Figure 8).

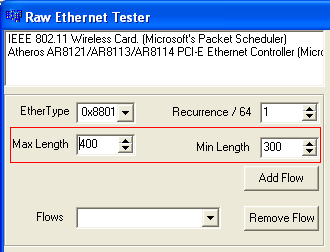


Figure 8: Max-Min selection for flow packet payload lengths

* Thirdly, each flow has a packet generation character set by its recurrence value over at most 64 slots in one generation cycle (Figure 9).

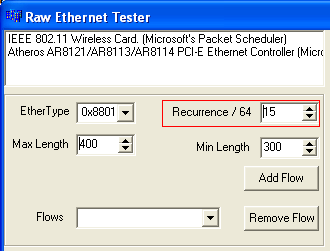


Figure 9: Recurrence selection for flow

* And finally, with the user defined flow parameters, the flow is created by the “Add Flow” button (Figure 10).

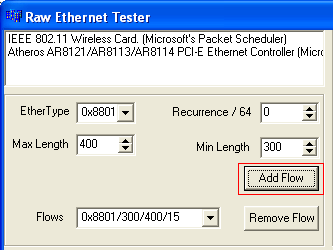


Figure 10: Adding a flow

The following figure (Figure 11) is showing an instance of generated flows on the application. There are 4 flows defined by ethertype/minPayloadLength/maxPayloadLength/recurrence. As we add all recurrence values (15,12,36 and 1) it makes 64. So when user starts a traffic generation a scenario of repetitive 64 slot cycles allocated by 4 flows will be generated first. The flow allocations will be spanning whole cycle randomly (Figure 11).

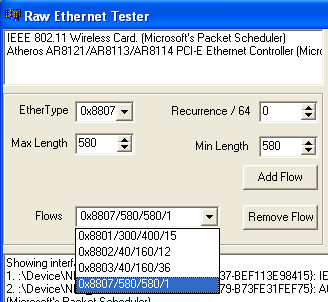


Figure 11: Created flows seen on the dropdown combobox

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| F1 | F4 | F1 | F3 | F2 | F2 | F3 | F2 |
| F3 | F1 | F3 | F2 | F3 | F1 | F3 | F3 |
| F3 | F2 | F3 | F3 | F1 | F3 | F3 | F3 |
| F1 | F3 | F2 | F1 | F3 | F2 | F1 | F2 |
| F3 | F2 | F1 | F3 | F1 | F3 | F3 | F3 |
| F3 | F3 | F3 | F1 | F3 | F1 | F3 | F1 |
| F3 | F3 | F3 | F2 | F3 | F3 | F3 | F1 |
| F2 | F3 | F3 | F1 | F2 | F3 | F3 | F3 |

Table 2: A pre-allocated cycle

The created flows can be removed from the scenario by first selecting the flow from the flows dropdown combobox and then clicking the “Remove Flow” button (Figure 12).

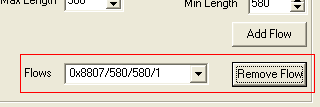


Figure 12: Removing a flow

Traffic generation is triggered by first double clicking on the desired output interface listed on the left hand side corner, secondly selection of number of cycles through a selector and finally clicking on the “Send” button (Figure 13, Figure 14).

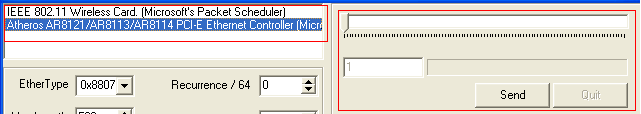


Figure 13: Starting packet generation

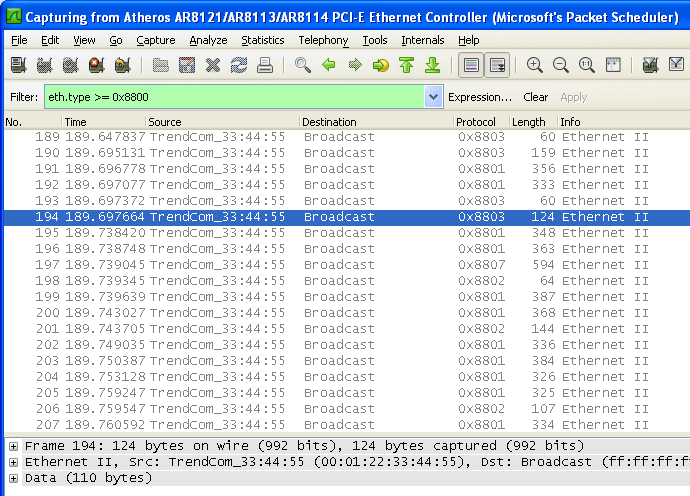


Figure 14: Packets generated as seen on Wireshark

This application uses the WinPcap library for transmission of ethernet packets through the selected Windows network interface. Thus, WinPcap driver should be installed before using the application. A fresh install of latest Wireshark Network Protocol Analyzer installs the WinPcap drivers on Windows.

# Emulation Test Setup

The test setup includes 2 computers:

* A single board computer running Ubuntu 10.10 Maverick and the QoS scheduler application. The SBC is an Advantech 9562 with Intel D510 Dual Core 1.66 GHz CPU, 2 GB ram and 3 ethernet ports. QoS scheduler application uses Eth0 as ingress port and Eth1 as egress port.



Figure 15: QoS running on SBC

* A test PC running Windows XP, Raw Ethernet Tester Application and Wireshark network analyzer.

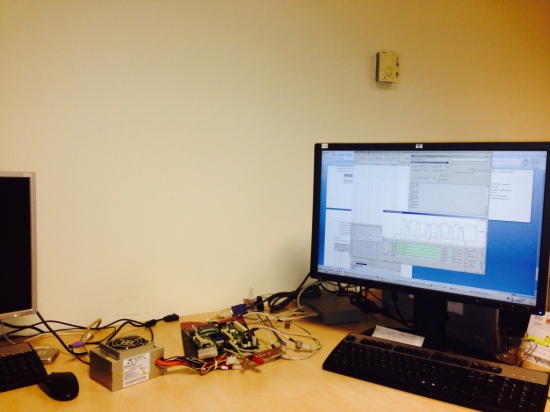


Figure 16: Test PC

Two computers are connected with each other via a 100 Mbit ethernet switch. The test PC is connected to the switch by its unique ethernet port. On the other hand, SBC is connected to the switch by its eth0 and eth1 ports (Figure 17).

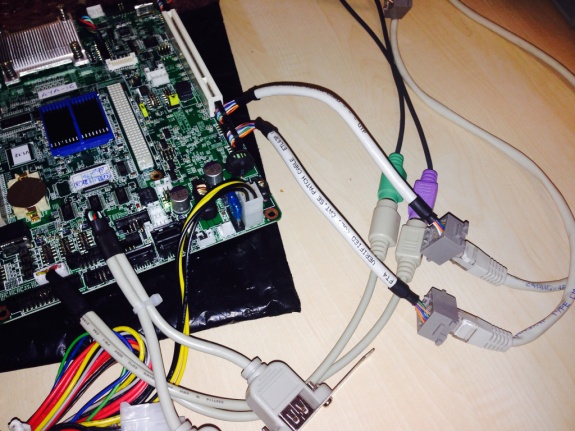


Figure 17: Eth0 and Eth1 ports of SBC

A typical test flow runs as follows:

* A Wireshark instance is configured to sniff packets on the ethernet and an IO graph is configured with flow identifiers, i.e. the ingress and egress ethertypes.
* User creates flows and starts a transmission from the raw ethernet tester application.
* User configures and starts the QoS scheduler application on the SBC.
* Raw ethernet tester application transmits ethernet packets for flows continuously.
* QoS scheduler application gets ingress flows from the eth0 port, schedules flows and transmits scheduled packets on the eth1 port.
* Wireshark application sniffs all the packets raw ethernet tester application has sent out, and the scheduled ethernet packets received from the SBC.
* A run for each scheduler is evaluated by using the Wireshark IO graph and QoS Scheduler statistical information dumped on the screen. On the other hand, Linux system monitor tool is also followed for gathering information about the schedulers processing power consumption.

## Test 1

This test was made to inspect the fairness of the schedulers for 3 flows, each generating same length packets with same transmission rate (Figure 18).

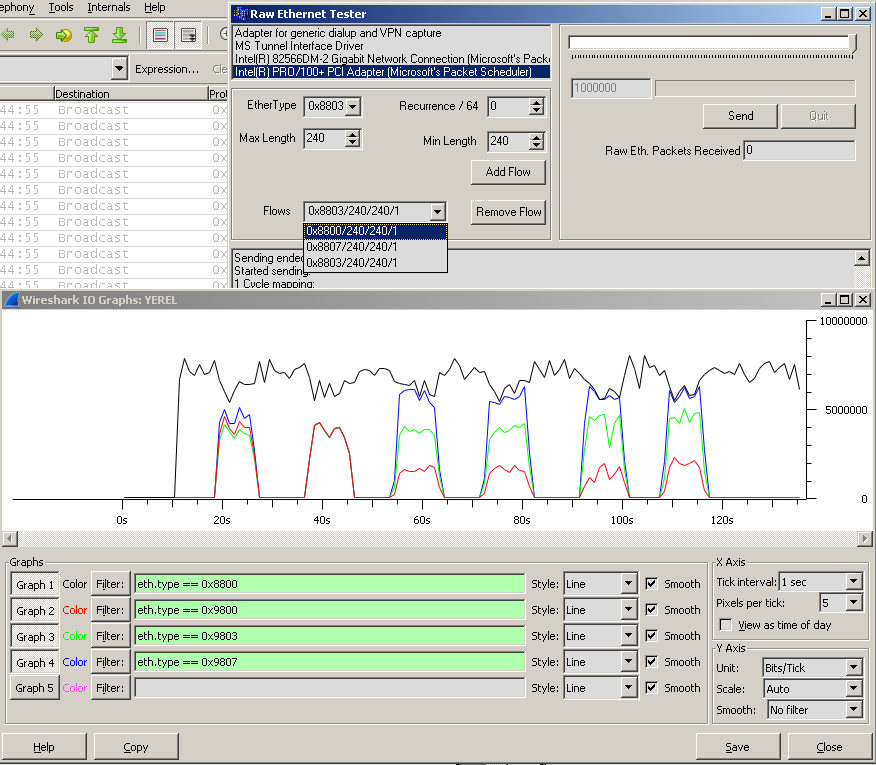


Figure 18: Test 1 setup and results on windows

The schedulers on the SBC have been started and stopped one by one on the Linux machine. Thus the IO graph on Figure 18 reflects a continuous transmission of a 0x8800 flow and particular flow graphs for 0x9800, 0x9803 and 0x9807 scheduling by FIFO, RR, WRR, DRR, WF2Q and DTable in order of appearance. The important point in this test is that the ingress link capacity is much more than the egress link capacity. Thus the fairness can be tested, while there are overflows on each flow queue.

As it is seen on the raw ethernet tester instance (Figure 18) there are 3 flows generated:

* 0x8800 flow with constant 240 bytes payload and recurrence of 1 in a cycle.
* 0x8803 flow with constant 240 bytes payload and recurrence of 1 in a cycle.
* 0x8807 flow with constant 240 bytes payload and recurrence of 1 in a cycle.

As all flows are of same characteristics, the IO graph on the Wireshark has been configured to show only the 0x8800 flow (black line) as a reference. The other flows configured are 0x9800 (red line), 0x9803 (green line) and 0x9807 (blue line). The ingress link rate can be calculated by multiplying the 0x8800 flow rate by 3, which results in roughly 21 Mbits/sec.

The scheduler behaviors are as follows:

* FIFO (17 to 27 secs. on timeline) schedules all flows as it appears on the ingress line and queued on the FIFO queue. There are some inequalities between 3 flows here, which have been randomly created because of the FIFO queue overflow just before scheduling. It should be recalled that the only one queue in this scheduler is prone to overflows more than other schedulers where there are separate queues for each flow. Total egress link rate here is roughly 15 Mbits/sec. Maximum and average queuing delay for the only flow is as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| All Flows as one flow | 96 msecs | 48.6363 msecs | 16171/4107434 | CPU1: 100%  CPU2: 66.3% |

Table 3: Test 1 FIFO statistics

* Round Robin scheduler (36 to 46 secs. on timeline) has given same amount of bandwidth to each flow appearing on the ingress link. What is better than FIFO is that the burstiness in the traffic is divided in between 3 different queues for each flow and thus each egress flow characteristics are just same. Total egress link rate here is roughly 15 Mbits/sec. Maximum and average queuing delays for the flows are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 44 msecs | 15.1762 msecs | 9283/2357882 | CPU1: 41.6%  CPU2: 30.8% |
| 0x8803 | 44 msecs | 15.1956 msecs | 9294/2360676 |
| 0x8807 | 44 msecs | 15.2141 msecs | 9296/2361184 |

Table 4: Test 1 RR statistics

* Weighted Round Robin scheduler (53 to 65 secs. on timeline) is fair among the flows as it gives the least bandwidth to the 0x8800 flow and the most bandwidth to the 0x8807 flow. As the packet lengths for each flow are the same, it is no surprise that WRR resulted in the same fairness with respect to DRR, WF2Q and D table. Total egress link rate here is roughly 15 Mbits/sec. Maximum and average queuing delays for the flows are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 116 msecs | 40.9257 msecs | 19436/4936744 | CPU1: 39.6%  CPU2: 26.8% |
| 0x8803 | 52 msecs | 15.1457 msecs | 10426/2648204 |
| 0x8807 | 36 msecs | 5.89809 msecs | 2132/541528 |

Table 5: Test 1 WRR statistics

* Deficit Round Robin (70 to 82 secs. on timeline) scheduler is fair among the flows. Total egress link rate here is roughly 15 Mbits/sec. Total egress link rate here is roughly 15 Mbits/sec. Maximum and average queuing delays for the flows are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 120 msecs | 41.1778 msecs | 19553/4966462 | CPU1: 41%  CPU2: 34% |
| 0x8803 | 56 msecs | 14.4743 msecs | 9974/2533396 |
| 0x8807 | 44 msecs | 5.55282 msecs | 2444/620776 |

Table 6: Test 1 DRR statistics

* WF2Q (91 to 102 sec. on timeline) scheduler is fair among the flows. Total egress link rate here is roughly 15 Mbits/sec. Total egress link rate here is roughly 15 Mbits/sec. Maximum and average queuing delays for the flows are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 196 msecs | 44.0645 msecs | 18316/4652264 | CPU1: 84.3%  CPU2: 51.1% |
| 0x8803 | 96 msecs | 11.2715 msecs | 7031/1785874 |
| 0x8807 | 32 msecs | 3.49714 msecs | 999/253746 |

Table 7: Test 1 WF2Q statistics

* DTable (107 to 117 sec. on timeline) scheduler is fair among the flows. Total egress link rate here is roughly 15 Mbits/sec. Total egress link rate here is roughly 15 Mbits/sec. Maximum and average queuing delays for the flows are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 128 msecs | 34.1573 msecs | 16424/4171696 | CPU1: 41.6%  CPU2: 30.8% |
| 0x8803 | 40 msecs | 11.4939 msecs | 6432/1633728 |
| 0x8807 | 24 msecs | 3.94444 msecs | 1039/263906 |

Table 8: Test 1 DTable statistics

This test shows that the most successful scheduler regarding the queuing delays is the WF2Q scheduler, where the nearest performance is made by the DTable scheduler. DTable scheduler is also showing really better processing power consumption than the WF2Q scheduler. Thus, this test approves the delay related founding of the authors in the paper.

## Test 2

This test is a repetition of Test 1 with egress link rate higher than the total flow input rate. Thus, all flows are expected to be scheduled without any overflows in the queues and packet losses.

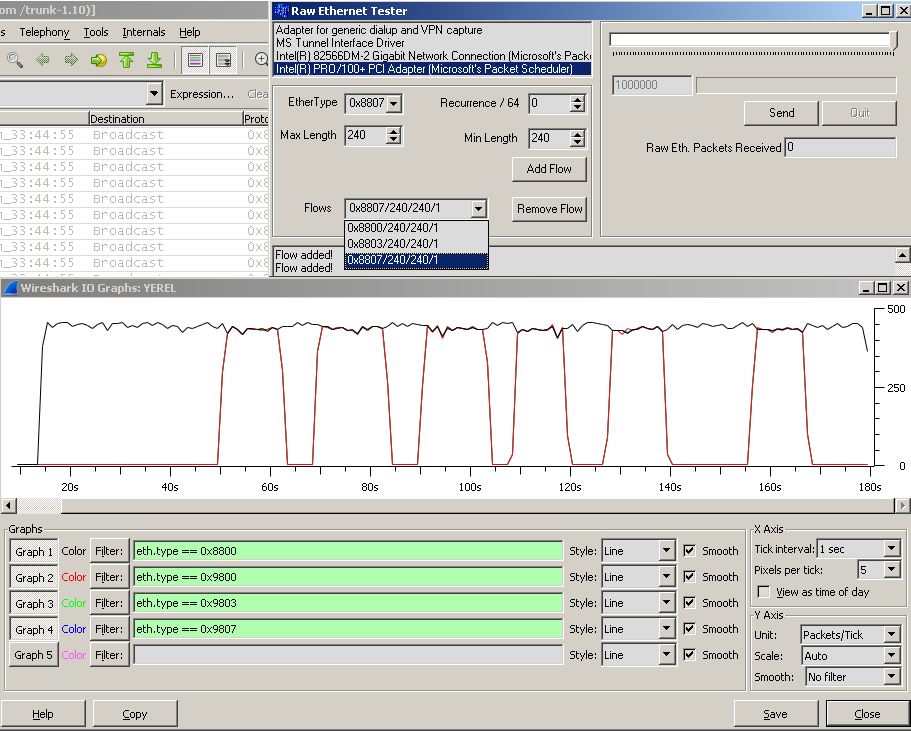


Figure 19: Test 2 setup and results on windows

The scheduler behaviors are as follows:

* FIFO (50 to 63 secs. on timeline) schedules all flows as it appears on the ingress line and queued on the FIFO queue. Total egress link rate here is roughly 2.4 Mbits/sec. Maximum and average queuing delay for the only flow is as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| All Flows as one flow | 20 msecs | 1.71909 msecs | 0/0 | CPU1: 71.7%  CPU2: 60.6% |

Table 9: Test 2 FIFO statistics

* Round Robin scheduler (68 to 84 secs. on timeline) has given same amount of bandwidth to each flow appearing on the ingress link. Total egress link rate here is roughly 2.4 Mbits/sec. Maximum and average queuing delays for the flows are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 20 msecs | 1.87806 msecs | 0/0 | CPU1: 75%  CPU2: 46.5% |
| 0x8803 | 20 msecs | 1.89342 msecs | 0/0 |
| 0x8807 | 20 msecs | 1.94335 msecs | 0/0 |

Table 10: Test 2 RR statistics

* Weighted Round Robin scheduler (90 to 104 secs. on timeline) scheduler has given same amount of bandwidth to each flow appearing on the ingress link. Total egress link rate here is roughly 2.4 Mbits/sec. Maximum and average queuing delays for the flows are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 36 msecs | 3.45766 msecs | 0/0 | CPU1: 54.5%  CPU2: 72.2% |
| 0x8803 | 24 msecs | 1.82402 msecs | 0/0 |
| 0x8807 | 16 msecs | 1.51785 msecs | 0/0 |

Table 11: Test 2 WRR statistics

* Deficit Round Robin (107 to 120 secs. on timeline) scheduler has given same amount of bandwidth to each flow appearing on the ingress link. Total egress link rate here is roughly 2.4 Mbits/sec. Maximum and average queuing delays for the flows are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 60 msecs | 3.10506 msecs | 0/0 | CPU1: 46.4%  CPU2: 93.6% |
| 0x8803 | 28 msecs | 1.63374 msecs | 0/0 |
| 0x8807 | 16 msecs | 1.26068 msecs | 0/0 |

Table 12: Test 2 DRR statistics

* WF2Q (126 to 140 sec. on timeline) scheduler has given same amount of bandwidth to each flow appearing on the ingress link. Total egress link rate here is roughly 2.4 Mbits/sec. Maximum and average queuing delays for the flows are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 32 msecs | 3.44619 msecs | 0/0 | CPU1: 87.9%  CPU2: 42.2% |
| 0x8803 | 24 msecs | 1.90637 msecs | 0/0 |
| 0x8807 | 16 msecs | 1.40663 msecs | 0/0 |

Table 13: Test 2 WF2Q statistics

* DTable (155 to 168 sec. on timeline) scheduler has given same amount of bandwidth to each flow appearing on the ingress link. Total egress link rate here is roughly 2.4 Mbits/sec. Maximum and average queuing delays for the flows are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 28 msecs | 2.97281 msecs | 0/0 | CPU1: 63.6%  CPU2: 75.2% |
| 0x8803 | 20 msecs | 1.63874 msecs | 0/0 |
| 0x8807 | 16 msecs | 1.32048 msecs | 0/0 |

Table 14: Test 2 DTable statistics

This test shows that all schedulers have done well on fixed size flows with enough bandwidth given on the egress link. As there has been no bursty traffic on timeline, and advantage of the DTable scheduler has not been faced with.

## Test 3

This test is a modified version of test 1 where flow 0 is configured with bigger packet size and the egress link is limited to 2 Mbits/sec rate (Figure 20). The traffic rate in the ingress link is higher than the egress link rate here. Thus, packet losses are expected in the flows.

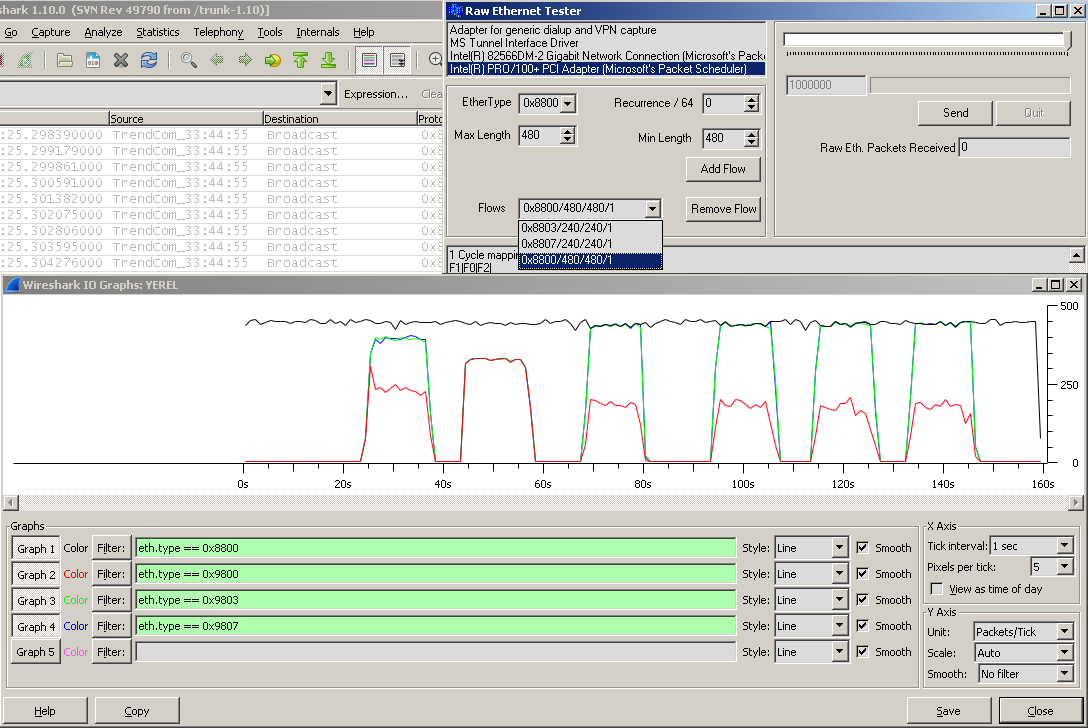


Figure 20: Test 3 setup and results on windows

As it is seen on the raw ethernet tester instance (Figure 20) there are 3 flows generated:

* 0x8800 flow with constant **480** bytes payload and recurrence of 1 in a cycle.
* 0x8803 flow with constant 240 bytes payload and recurrence of 1 in a cycle.
* 0x8807 flow with constant 240 bytes payload and recurrence of 1 in a cycle.

The statistical values realized are given in the following tables:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| All Flows as one flow | 280 msecs | 245.802 msecs | 3371/79946 | CPU1: 31.3%  CPU2: 30 |

Table 15: Test 3 FIFO statistics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 76 msecs | 60.6697 msecs | 1373/678262 | CPU1: 37.9%  CPU2: 71.2% |
| 0x8803 | 136 msecs | 117.119 msecs | 1355/344170 |
| 0x8807 | 136 msecs | 117.16 msecs | 1355/344170 |

Table 16: Test 3 RR statistics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 192 msecs | 104.083 msecs | 2627/1297738 | CPU1: 36.6%  CPU2: 72.3% |
| 0x8803 | 40 msecs | 3.88276 msecs | 0/0 |
| 0x8807 | 12 msecs | 1.95943 msecs | 0/0 |

Table 17: Test 3 WRR statistics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 168 msecs | 105.317 msecs | 2738/1352572 | CPU1: 93.9%  CPU2: 56.5% |
| 0x8803 | 20 msecs | 2.17998 msecs | 0/0 |
| 0x8807 | 16 msecs | 1.60892 msecs | 0/0 |

Table 18: Test 3 DRR statistics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 240 msecs | 114.359 msecs | 2893/1429142 | CPU1: 78.2%  CPU2: 51.0% |
| 0x8803 | 24 msecs | 4.09684 msecs | 0/0 |
| 0x8807 | 20 msecs | 2.04252 msecs | 0/0 |

Table 19: Test 3 WF2Q statistics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 176 msecs | 107.558 msecs | 2740/1353560 | CPU1: 75.0%  CPU2: 39.0% |
| 0x8803 | 28 msecs | 2.3184 msecs | 0/0 |
| 0x8807 | 12 msecs | 1.61604 msecs | 0/0 |

Table 20: Test 3 DTable statistics

As the egress link rate is less than the injected traffic rate on the ingress link, there have been packet losses in

## Test 4

Test 4 is same as test 3 except that the egress link rate is limited to 1 Mbits/sec. This time the egress link rate is much less than the traffic rate on the ingress link. More packet losses and queue overflows than that of test 3 are expected here.

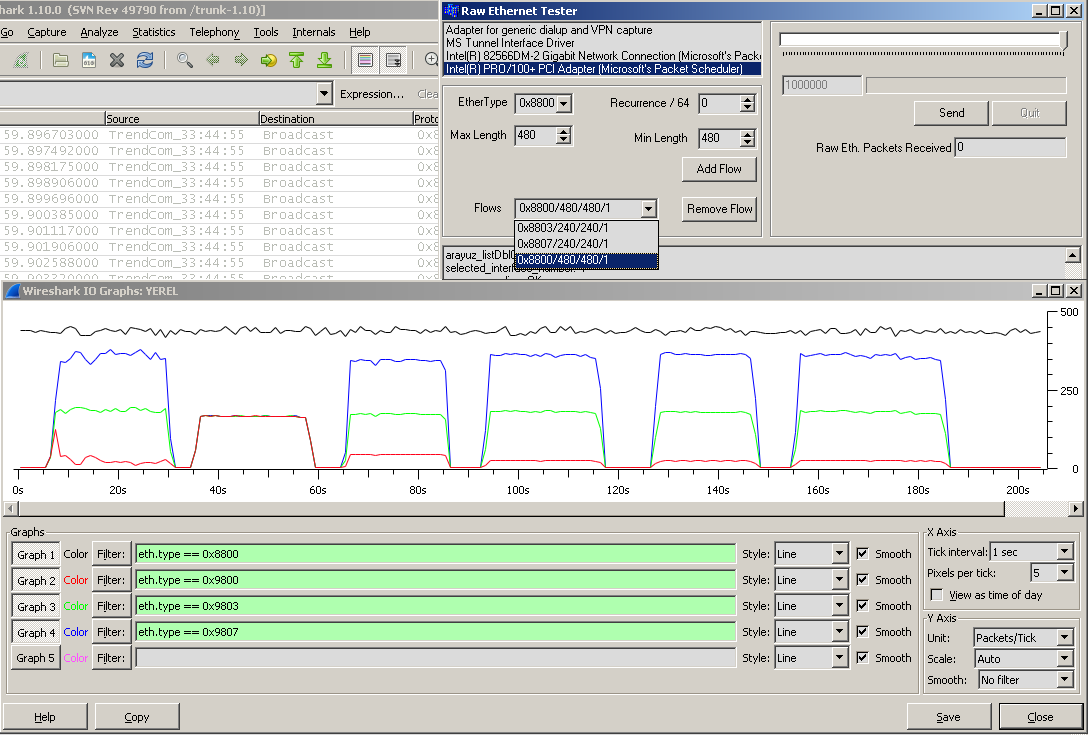


Figure 21: Test 4 setup and results on windows

The statistical values realized are given in the following tables:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| All Flows as one flow | 564 msecs | 524.959 msecs | 10944/5512496 | CPU1: 29.3%  CPU2: 56.4 |

* Table 21: Test 4 FIFO statistics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 140 msecs | 121.7 msecs | 5880/2904720 | CPU1: 40.6%  CPU2: 54.4% |
| 0x8803 | 264 msecs | 234.479 msecs | 5860/1488440 |
| 0x8807 | 260 msecs | 234.532 msecs | 5860/1488440 |

* Table 22: Test 4 RR statistics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 504 msecs | 463.03 msecs | 7641/3774654 | CPU1: 45.2%  CPU2: 76.7% |
| 0x8803 | 260 msecs | 226.353 msecs | 5139/1305306 |
| 0x8807 | 140 msecs | 111.891 msecs | 1826/463804 |

* Table 23: Test 4 WRR statistics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 888 msecs | 847.244 msecs | 8470/4184180 | CPU1: 54.7%  CPU2: 79.4% |
| 0x8803 | 240 msecs | 214.786 msecs | 5217/1325118 |
| 0x8807 | 132 msecs | 106.179 msecs | 9906/383032 |

* Table 24: Test 4 DRR statistics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 904 msecs | 859.916 msecs | 7805/3855670 | CPU1: 45.0%  CPU2: 59.4% |
| 0x8803 | 236 msecs | 217.292 msecs | 4865/1235710 |
| 0x8807 | 124 msecs | 105.602 msecs | 1408/357632 |

* Table 25: Test 4 WF2Q statistics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow No** | **Maximum Queuing Delay** | **Average Queuing Delay** | **Queue Over Flow in Packets/Bytes** | **Processing Power Consumption** |
| 0x8800 | 912 msecs | 840.359 msecs | 8442/4170348 | CPU1: 70.3%  CPU2: 34.7% |
| 0x8803 | 240 msecs | 214.431 msecs | 5197/1320038 |
| 0x8807 | 128 msecs | 105.271 msecs | 1496/379984 |

* Table 26: Test 4 DTable statistics

This test has shown no advantage of the DTable scheduler over DRR where they have both given the similar latency and fairness results and consumed less than WF2Q.

# Results

The project has been started on making a comparison between the simulation results from paper “Providing QoS with the Deficit Table Scheduler” and the emulation setup prepared. Tests made on the project have been totally different than the test made on the authors’ simulation work because of lack of infrastructural capabilities.

The results have shown that there has been no particular improvement in the latency performance with the DTable scheduler and the latency-bandwidth correlation couldn’t have been broken except Test 1. But it should be recalled that the authors are also proposing an arbitration mechanism to increase the performance of the DTable scheduler. Here, tests in this project have been done with a very similar slot assignment to that of WRR scheduler. The schedulers can be examined from the source code to see DTable implementations similarities to the DRR and WRR implementations.

To sum up, as an arbitration scheme experiment was not made, DTable scheduler has not shown any clue to make a difference in latency performance when there are flows of priorities far away from each other. Work done here in this project has not reached authors’ environment and simulation capabilities. Thus, this work should just be taken as a good startup to make research in this area.