The IP Differentiated Services Code Point (DSCP)

The IP Differentiated Services Code Point (DSCP)

The Differentiated Services framework is defined in RFC 2474 and RFC 2475. This defines a set of network policies and rules for a network domain. A router reads the DSCP value in each received packet to classify the packet. Once the class has been determined, it is mapped to one of 64 possible forwarding behaviors known as Per Hop Behavior group (PHB group). Multiple DSCPs can be mapped to the same PHB group, but each PHB group has a queue in which the packets are stored prior to forwarding.

Each PHB provides a particular service level (capacity, queuing, and dropping decisions) in accordance with a network policy. The AF group allows packets to be preferentially discarded when a particular AF class is overloaded.

- At the edges of a Differentiated Services domain, packets can be conditioned. That is they can be policed (dropped according to a policy), remarked (assigned to a different DSCP) or shaped (delayed with respect to other traffic). This happens at the network edge to enforce the required treatment across the domain. For example, a packet with an EF DSCP can be admission-controlled to protect the network from overload by excessive EF traffic. The DSCP value CS6 and CS7 could also be blocked at the edge, when an operator uses these classes for network control traffic.
- Measurements in 2017 suggest it is safe to enable DSCP in enedpoints by setting a suitable value in the IP header. This is based on observations that packets are seldom dropped solely because a non-default DSCP was set. Applications though need to be aware that the DSCP they choose may (or may not) have an assigned PHB - that is the network could decide to ignore the DSCP value, or even to rest the field to another value.

- The 6bit DSCP is appended to the header of an IPv4 or IPv6 packet in order to ensure the packet receives a particular forwarding treatment as it traverses a DiffServ enabled IP network. Different DSCP values will be used to apply different forwarding treatment and as such, a packet must be classified before ingress to the network to make sure that packets are given the appropriate forwarding treatment.

TCP/IP Model

TCP/IP Model

- Prerequisite – Layers of OSI Model The **OSI Model** we just looked at is just a reference/logical model. It was designed to describe the functions of the communication system by dividing the communication procedure into smaller and simpler components. But when we talk about the TCP/IP model, it was designed and developed by Department of Defense (DoD) in 1960s and is based on standard protocols. It stands for Transmission Control Protocol/Internet Protocol. The **TCP/IP model** is a concise version of the OSI model. It contains four layers, unlike seven layers in the OSI model. The layers are:
- Process/Application Layer
- Host-to-Host/Transport Layer
- Internet Layer
- Network Access/Link Layer
- What are the main 5 layers in TCP IP model?
- **Each host that is involved in a communication transaction runs a unique implementation of the protocol stack.**
- Physical Network Layer. The physical network layer specifies the characteristics of the hardware to be used for the network. ...
- Data-Link Layer. ...
- Internet Layer. ...
- Transport Layer. ...
- Application Layer.

TCP/IP	OSI
TCP refers to Transmission Control Protocol.	OSI refers to Open Systems Interconnection.
TCP/IP has 4 layers.	OSI has 7 layers.
TCP/IP is more reliable	OSI is less reliable
TCP/IP does not have very strict boundaries.	OSI has strict boundaries
TCP/IP follow a horizontal approach.	OSI follows a vertical approach.
TCP/IP uses both session and presentation layer in the application layer itself.	OSI uses different session and presentation layers.
TCP/IP developed protocols then model.	OSI developed model then protocol.
Transport layer in TCP/IP does not provide assurance delivery of packets.	In OSI model, transport layer provides assurance delivery of packets.
TCP/IP model network layer only provides connection less services.	Connection less and connection oriented both services are provided by network layer in OSI model.
Protocols cannot be replaced easily in TCP/IP model.	While in OSI model, Protocols are better covered and is easy to replace with the change in technology.

- 1. Network Access Layer –

- This layer corresponds to the combination of Data Link Layer and Physical Layer of the OSI model. It looks out for hardware addressing and the protocols present in this layer allows for the physical transmission of data. We just talked about ARP being a protocol of Internet layer, but there is a conflict about declaring it as a protocol of Internet Layer or Network access layer. It is described as residing in layer 3, being encapsulated by layer 2 protocols.

- The Network Access Layer is a layer in the OSI model that is responsible for establishing a connection between a device and the physical network. It is responsible for transmitting and receiving data over the physical medium of the network, such as a wire or wireless connection.

- One common use case of the Network Access Layer is in networking devices, such as routers and switches. These devices use the Network Access Layer to establish connections with other devices on the network and transmit and receive data. For example, a router may use the Network Access Layer to establish a connection with a device on the network and then forward data packets to and from that device.

- Another use case of the Network Access Layer is in communication devices, such as phones and laptops. These devices use the Network Access Layer to establish a connection with a wireless or wired network and transmit and receive data over that connection. For example, a phone may use the Network Access Layer to connect to a wireless network and send and receive phone calls and text messages.

- **2. Internet Layer -**
- This layer parallels the functions of OSI's Network layer. It defines the protocols which are responsible for logical transmission of data over the entire network. The main protocols residing at this layer are :
- **IP** - stands for Internet Protocol and it is responsible for delivering packets from the source host to the destination host by looking at the IP addresses in the packet headers. IP has 2 versions: IPv4 and IPv6. IPv4 is the one that most of the websites are using currently. But IPv6 is growing as the number of IPv4 addresses are limited in number when compared to the number of users.
- **ICMP** - stands for Internet Control Message Protocol. It is encapsulated within IP datagrams and is responsible for providing hosts with information about network problems.
- **ARP** - stands for Address Resolution Protocol. Its job is to find the hardware address of a host from a known IP address. ARP has several types: Reverse ARP, Proxy ARP, Gratuitous ARP and Inverse ARP.
- The Internet Layer is a layer in the Internet Protocol (IP) suite, which is the set of protocols that define the Internet. The Internet Layer is responsible for routing packets of data from one device to another across a network. It does this by assigning each device a unique IP address, which is used to identify the device and determine the route that packets should take to reach it.
- Here is an example of a use case for the Internet Layer:
- Imagine that you are using a computer to send an email to a friend. When you click "send," the email is broken down into smaller packets of data, which are then sent to the Internet Layer for routing. The Internet Layer assigns an IP address to each packet and uses routing tables to determine the best route for the packet to take to reach its destination. The packet is then forwarded to the next hop on its route until it reaches its destination. When all of the packets have been delivered, your friend's computer can reassemble them into the original email message.
- In this example, the Internet Layer plays a crucial role in delivering the email from your computer to your friend's computer. It uses IP addresses and routing tables to determine the best route for the packets to take, and it ensures that the packets are delivered to the correct destination. Without the Internet Layer, it would not be possible to send data across the Internet.

- **3. Host-to-Host Layer -**
- This layer is analogous to the transport layer of the OSI model. It is responsible for end-to-end communication and error-free delivery of data. It shields the upper-layer applications from the complexities of data. The two main protocols present in this layer are :
- **Transmission Control Protocol (TCP) -** It is known to provide reliable and error-free communication between end systems. It performs sequencing and segmentation of data. It also has acknowledgment feature and controls the flow of the data through flow control mechanism. It is a very effective protocol but has a lot of overhead due to such features. Increased overhead leads to increased cost.
- **User Datagram Protocol (UDP) -** On the other hand does not provide any such features. It is the go-to protocol if your application does not require reliable transport as it is very cost-effective. Unlike TCP, which is connection-oriented protocol, UDP is connectionless.
  - **HTTP and HTTPS -** HTTP stands for Hypertext transfer protocol. It is used by the World Wide Web to manage communications between web browsers and servers. HTTPS stands for HTTP-Secure. It is a combination of HTTP with SSL(Secure Socket Layer). It is efficient in cases where the browser need to fill out forms, sign in, authenticate and carry out bank transactions.
  - **SSH -** SSH stands for Secure Shell. It is a terminal emulations software similar to Telnet. The reason SSH is more preferred is because of its ability to maintain the encrypted connection. It sets up a secure session over a TCP/IP connection.
  - **NTP -** NTP stands for Network Time Protocol. It is used to synchronize the clocks on our computer to one standard time source. It is very useful in situations like bank transactions. Assume the following situation without the presence of NTP. Suppose you carry out a transaction, where your computer reads the time at 2:30 PM while the server records it at 2:28 PM. The server can crash very badly if it's out of sync.
- The host-to-host layer is a layer in the OSI (Open Systems Interconnection) model that is responsible for providing communication between hosts (computers or other devices) on a network. It is also known as the transport layer.

User Datagram Protocol (UDP)

User Datagram Protocol (UDP)

- **User Datagram Protocol (UDP)** is a Transport Layer protocol. UDP is a part of the Internet Protocol suite, referred to as UDP/IP suite. Unlike TCP, it is an **unreliable and connectionless protocol.** So, there is no need to establish a connection prior to data transfer. The UDP helps to establish low-latency and loss-tolerating connections establish over the network.The UDP enables process to process communication.
- Though Transmission Control Protocol (TCP) is the dominant transport layer protocol used with most of the Internet services; provides assured delivery, reliability, and much more but all these services cost us additional overhead and latency. Here, UDP comes into the picture. For real-time services like computer gaming, voice or video communication, live conferences; we need UDP. Since high performance is needed, UDP permits packets to be dropped instead of processing delayed packets. There is no error checking in UDP, so it also saves bandwidth. User Datagram Protocol (UDP) is more efficient in terms of both latency and bandwidth.

- **UDP Header -**
- UDP header is an **8-bytes** fixed and simple header, while for TCP it may vary from 20 bytes to 60 bytes. The first 8 Bytes contains all necessary header information and the remaining part consist of data. UDP port number fields are each 16 bits long, therefore the range for port numbers is defined from 0 to 65535; port number 0 is reserved. Port numbers help to distinguish different user requests or processes.
- **Source Port:** Source Port is a 2 Byte long field used to identify the port number of the source.
- **Destination Port:** It is a 2 Byte long field, used to identify the port of the destined packet.
- **Length:** Length is the length of UDP including the header and the data. It is a 16-bits field.
- **Checksum:** Checksum is 2 Bytes long field. It is the 16-bit one's complement of the one's complement sum of the UDP header, the pseudo-header of information from the IP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets.

- **Notes** - Unlike TCP, the Checksum calculation is not mandatory in UDP. No Error control or flow control is provided by UDP. Hence UDP depends on IP and ICMP for error reporting. Also UDP provides port numbers so that is can differentiate between users requests.

- **Applications of UDP:**
- Used for simple request-response communication when the size of data is less and hence there is lesser concern about flow and error control.
- It is a suitable protocol for multicasting as UDP supports packet switching.
- UDP is used for some routing update protocols like RIP(Routing Information Protocol).
- Normally used for real-time applications which can not tolerate uneven delays between sections of a received message.
- The application layer can do some of the tasks through UDP-
  - Trace Route
  - Record Route
  - Timestamp
- UDP takes a datagram from Network Layer, attaches its header, and sends it to the user. So, it works fast.
- Actually, UDP is a null protocol if you remove the checksum field.
  - Reduce the requirement of computer resources.
  - When using the Multicast or Broadcast to transfer.
  - The transmission of Real-time packets, mainly in multimedia applications.

Work-let Name: Performance Evaluation of DSCP values in various network

Worklet Details

1. Worklet ID: CP214MS
2. College Name: Ramaiah Institute of Technology

KPIs achieved till now

Analysing through the resources provided which will be helpful in the further implementation of the project.

Any Challenges/ Issues faced

Impediment in learning about the packet analysis

Next Steps

Focusing on implementation of the concept

Key Achievements/ Outcome till now

First view on OSI model and structure'

Hands on experience with NS3 and Wireshark

Basic Network layer services

TCP,UDP,IP,headers parameters

Date: 24.01.2023

Work-let Name: Performance Evaluation of DSCP values in various network

Worklet Details

1. Worklet ID: CP214MS
2. College Name: Ramaiah Institute of Technology

Learnings :

1. **Understanding of each parameter in TCP,UDP,IP headers?**
2. **Hands-on experience with ns3.**
3. **Hands-on experience with wireshark.**

Date: 24.01.2023

Work-let Name: Performance Evaluation of DSCP values in various network

Worklet Details

1. Worklet ID: CP214MS
2. College Name: Ramaiah Institute of Technology

1.Understanding of each parameter in TCP,UDP,IP headers.

TCP/IP is a stream-oriented protocol, while UDP is a packet-oriented protocol. This means that TCP/IP is considered to be a long stream of data that is transmitted from one end of the connection to the other end, and another long stream of data flowing in the opposite direction.

TCP - used for traffic that we need all the data for. i.e HTML, pictures, etc. UDP - used for traffic that doesn't suffer much if a packet is dropped, i.e. video & voice streaming, some data channels of online games, etc.

Date: 24.01.2023

Work-let Name: Performance Evaluation of DSCP values in various network

Worklet Details

1. Worklet ID: CP214MS
2. College Name: Ramaiah Institute of Technology

1.Understanding of each parameter in TCP,UDP,IP headers.

TCP **provides communication between an application program and the Internet Protocol** (they are frequently written as TCP/IP.) An application does not need to required packet fragmentation on the transmission medium or other mechanisms for sending data in order to be sent via TCP.

Whereas

User datagram protocol (UDP) is used for **time-critical data transmissions such as DNS lookups, online gaming, and video streaming**. This communication protocol boosts transfer speeds by removing the need for a formal two-way connection before the data transmission begins.

Date: 24.01.2023

Worklet Details

1. Worklet ID: CP214MS
2. College Name: Ramaiah Institute of Technology

2. What is Ns3 ?

- ns-3 has been developed **to provide an open, extensible network simulation platform, for networking research and education**. In brief, ns-3 provides models of how packet data networks work and perform, and provides a simulation engine for users to conduct simulation experiments.
- We had a hand's on experience with ns-3 on linux.
- Learnt about basics of building topologies using ns-3.

Worklet Details

1. Worklet ID: CP214MS
2. College Name: Ramaiah Institute of Technology

2. What is Ns3 ?

- A few key points are worth noting at the onset:

ns-3 is open-source, and the project strives to maintain an open environment for researchers to contribute and share their software.

- *ns-3* is not a backwards-compatible extension of ns-2; it is a new simulator. The two simulators are both written in C++ but *ns-3* is a new simulator that does not support the *ns-2* APIs.

Date: 24.01.2023

Worklet Details

1. Worklet ID: CP214MS
2. College Name: Ramaiah Institute of Technology

3. What is WireShark ?

Wireshark is **a network protocol analyzer, or an application that captures packets from a network connection, such as from your computer to your home office or the internet**. Packet is the name given to a discrete unit of data in a typical Ethernet network. Wireshark is the most often-used packet sniffer in the world.

Work-let Name: Performance Evaluation of DSCP values in various network

Worklet Details

1. Worklet ID: CP214MS
2. College Name: Ramaiah Institute of Technology

3. What is WireShark ?

We tried analyzing some packets through WireShark for better learning of the concepts of packet flow and to get an overview of the software we have to use for the further project implementation.

Date: 24.01.2023

Differentiated Service Queuing Disciplines in NS-3

Robert Chang, Mahdi Rahimi, and Vahab Pournaghshband

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

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

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

I. INTRODUCTION

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

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

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

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

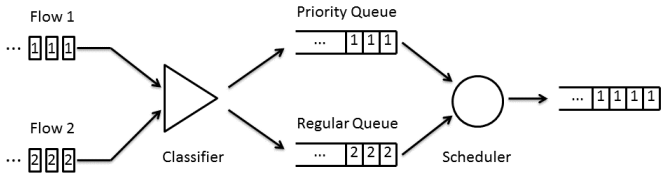


Figure 1. A strict priority queue

II. BACKGROUND

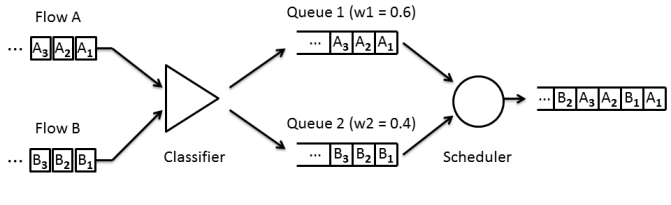
A. Strict Priority Queuing

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

B. Weighted Fair Queuing

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

 ISBN: 978-1-61208-442-8



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

Figure 2. A weighted fair queue

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

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

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

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

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

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

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

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

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

C. Weighted Round Robin

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

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

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

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

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

III. Related Work

The predecessor to `ns-3`, `ns-2` [9], had implemented some scheduling algorithms such fair queuing, stochastic fair queuing, smoothed round robin, deficit round robin, priority queuing, and class based queuing as official modules. `ns-2`and `ns-3` are essentially different and incompatible environments, `ns-3` is a new simulator written from scratch and is not an

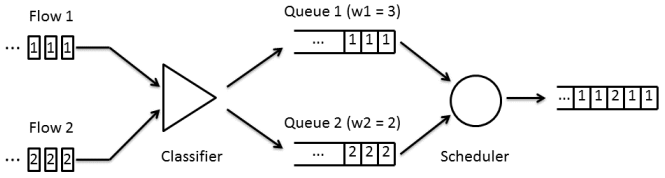


Figure 3. A weighted round robin queue

evolution of ns-2. At the time of writing, the latest version, ns-3.32, contains no official differentiated service queuing modules. Several modules have been contributed by others, such as a leaky bucket queue implementation [2] and the previously mentioned DiffServ evaluation module [3].

IV. Design and Implementation

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

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

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

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

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

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

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

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

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

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

A. Strict Priority Queuing

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

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

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

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

B. Weighted Fair Queuing

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

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

Therefore, there are mainly two events that impact $V(t)$: first, a packet arrival that is the time an inactive queue becomes active and second, when a queue finishes service and becomes

 ISBN: 978-1-61208-442-8

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

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

Figure 4. The iterated deletion algorithm

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

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

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

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

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

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

C. Weighted Round Robin

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

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

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

V. VALIDATION

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

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

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

 ISBN: 978-1-61208-442-8

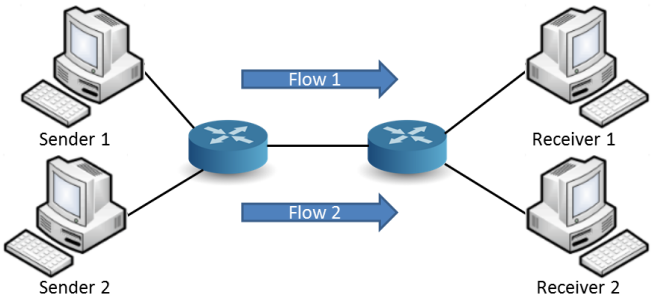


Figure 5. Simulated network used in validation experiments

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

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

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

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

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

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

VI. Usage

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

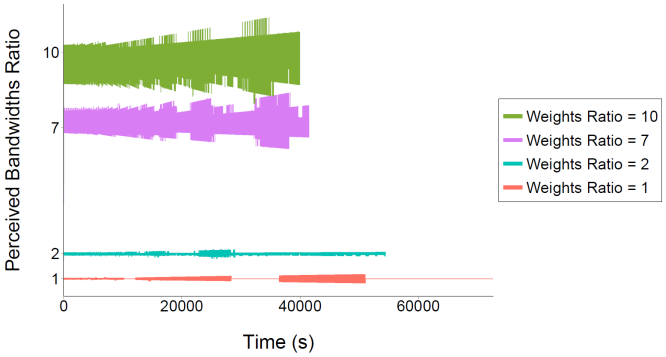


Figure 6. WFQ validation: T = 0.5 Mbps

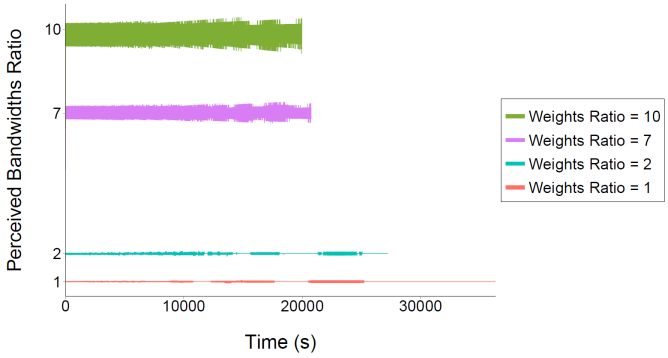


Figure 7. WFQ validation: T = 1 Mbps

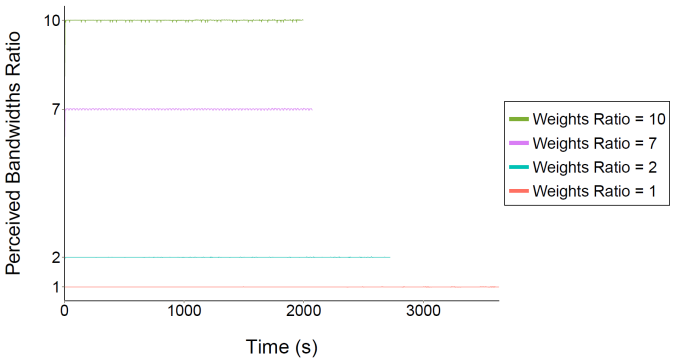


Figure 8. WFQ validation: T = 10 Mbps

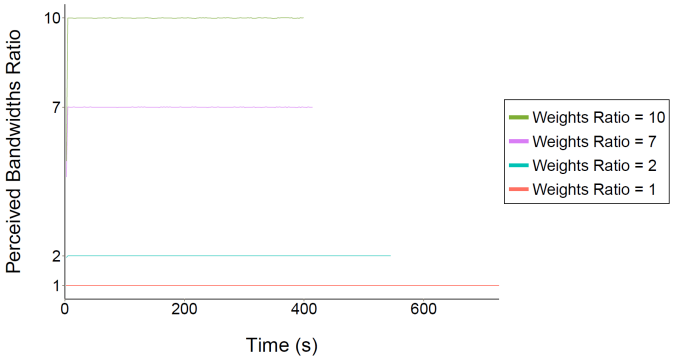


Figure 9. WFQ validation: T = 50 Mbps

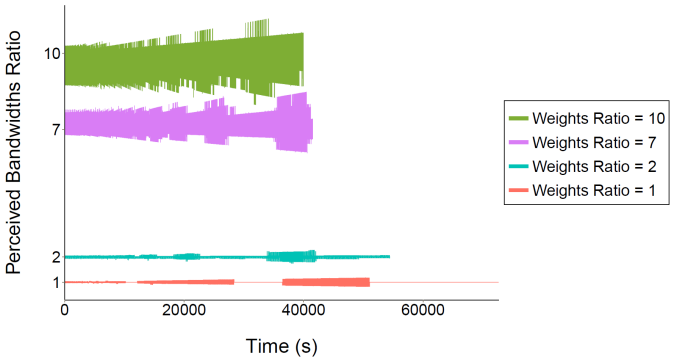


Figure 10. WRR validation: T = 0.5 Mbps

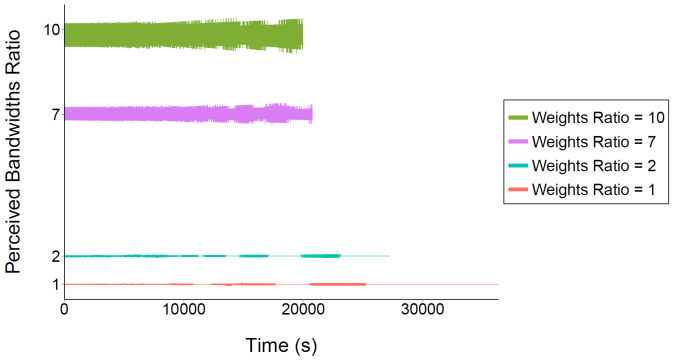


Figure 11. WRR validation: T = 1 Mbps

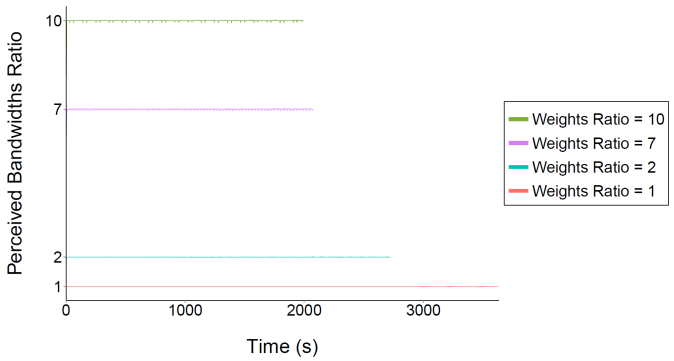


Figure 12. WRR validation: T = 10 Mbps

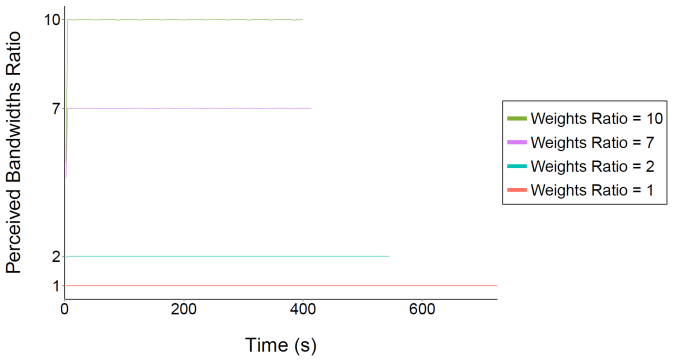


Figure 13. WRR validation: T = 50 Mbps

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

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

VII. Conclusion and Future Work

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

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

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

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

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

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

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

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

References

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

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

 ISBN: 978-1-61208-442-8

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

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

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

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

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

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

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

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

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

 ISBN: 978-1-61208-442-8

Differentiated Service Queuing Disciplines in NS-3

Robert Chang, Mahdi Rahimi, and Vahab Pournaghshband

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

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

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

I. INTRODUCTION

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

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

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

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

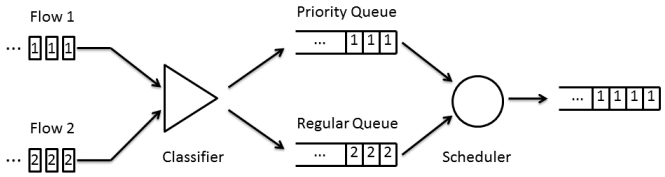


Figure 1. A strict priority queue

II. BACKGROUND

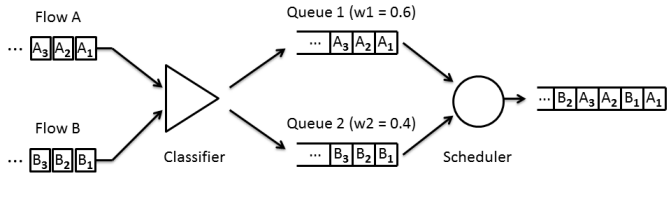
A. Strict Priority Queuing

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

B. Weighted Fair Queuing

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

 ISBN: 978-1-61208-442-8



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

Figure 2. A weighted fair queue

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

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

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

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

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

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

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

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

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

C. Weighted Round Robin

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

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

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

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

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

III. Related Work

The predecessor to `ns-3`, `ns-2` [9], had implemented some scheduling algorithms such fair queuing, stochastic fair queuing, smoothed round robin, deficit round robin, priority queuing, and class based queuing as official modules. `ns-2`and `ns-3` are essentially different and incompatible environments, `ns-3` is a new simulator written from scratch and is not an

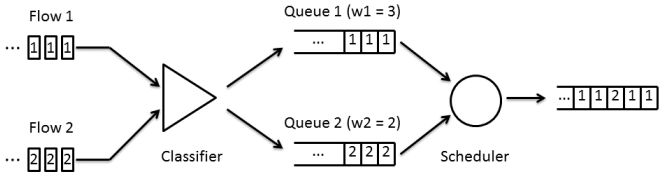


Figure 3. A weighted round robin queue

evolution of ns-2. At the time of writing, the latest version, ns-3.32, contains no official differentiated service queuing modules. Several modules have been contributed by others, such as a leaky bucket queue implementation [2] and the previously mentioned DiffServ evaluation module [3].

IV. Design and Implementation

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

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

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

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

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

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

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

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

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

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

A. Strict Priority Queuing

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

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

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

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

B. Weighted Fair Queuing

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

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

Therefore, there are mainly two events that impact $V(t)$: first, a packet arrival that is the time an inactive queue becomes active and second, when a queue finishes service and becomes

 ISBN: 978-1-61208-442-8

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

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

Figure 4. The iterated deletion algorithm

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

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

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

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

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

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

C. Weighted Round Robin

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

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

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

V. VALIDATION

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

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

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

 ISBN: 978-1-61208-442-8

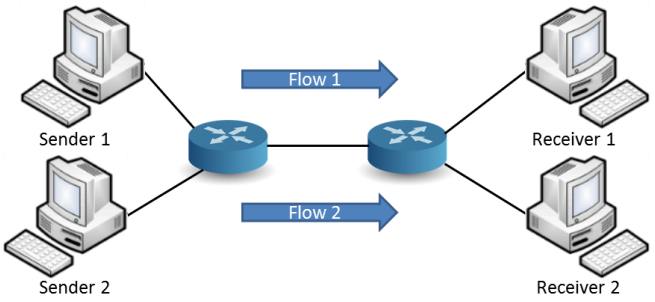


Figure 5. Simulated network used in validation experiments

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

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

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

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

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

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

VI. Usage

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

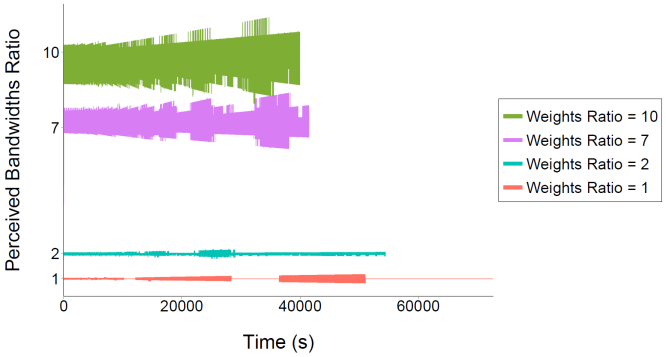


Figure 6. WFQ validation: T = 0.5 Mbps

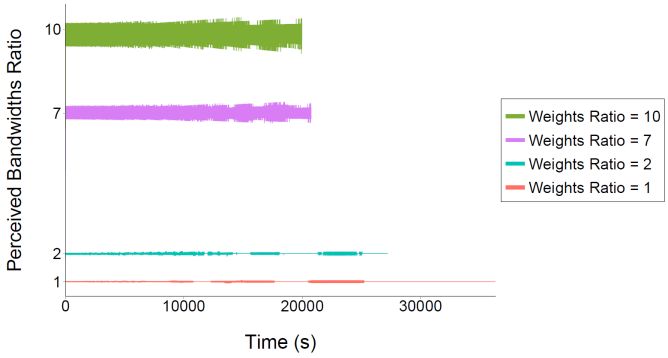


Figure 7. WFQ validation: T = 1 Mbps

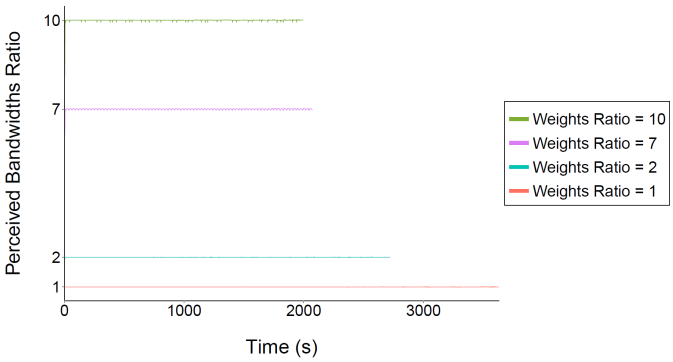


Figure 8. WFQ validation: T = 10 Mbps

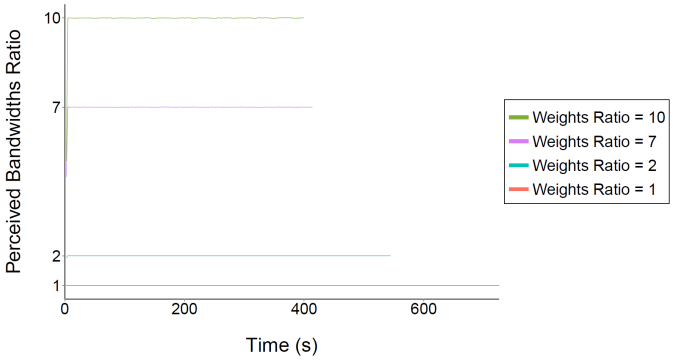


Figure 9. WFQ validation: T = 50 Mbps

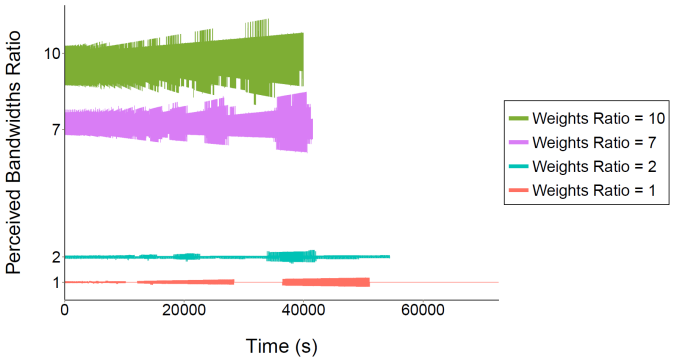


Figure 10. WRR validation: T = 0.5 Mbps

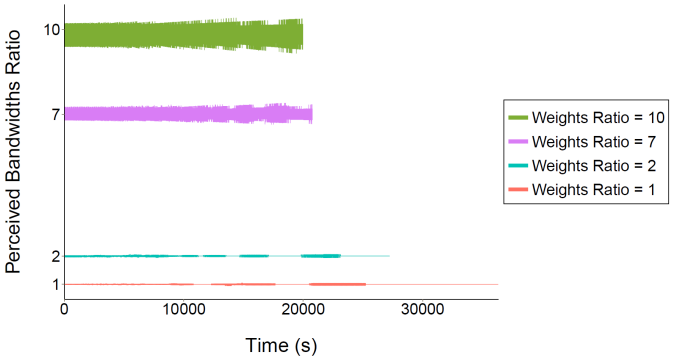


Figure 11. WRR validation: T = 1 Mbps

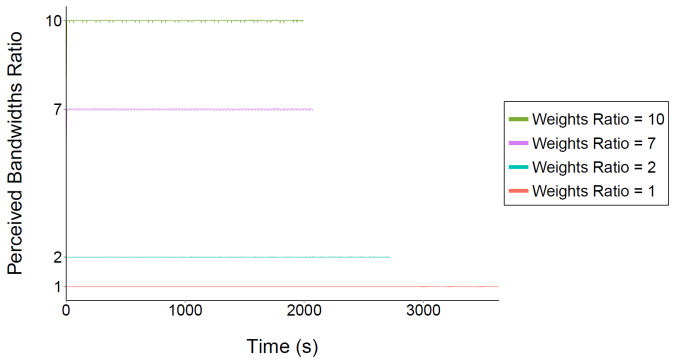


Figure 12. WRR validation: T = 10 Mbps

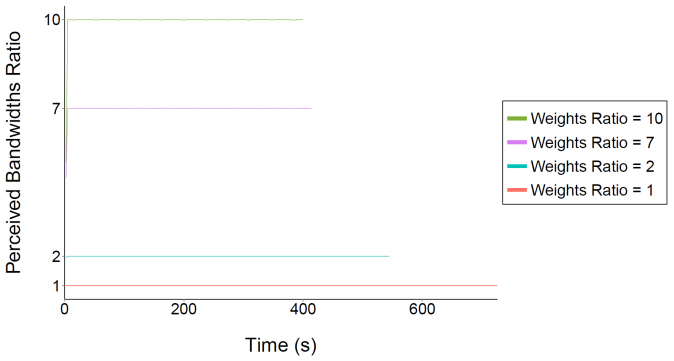


Figure 13. WRR validation: T = 50 Mbps

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

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

VII. Conclusion and Future Work

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

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

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

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

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

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

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

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

References

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

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

 ISBN: 978-1-61208-442-8

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

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

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

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

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

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

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

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

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

 ISBN: 978-1-61208-442-8

Work-let Name: Performance Evaluation of DSCP values in various network

Worklet Details

1. **Worklet ID:** CP214MS
2. **College Name: Ramaiah Institute of Technology.**

KPIs achieved till now

We have implemented different types of topologies in NS3 and analysed the traffic using Wire shark.
We learnt to configure client and server.

Any Challenges/ Issues faced

To simulate different network scenarios and also to set up different DSCP code points between client and the server.

Next Steps

Performance evaluation with different DSCP values in different network conditions like lossy,congested etc.

Key Achievements/ Outcome till now

We have setup different client and server and have implemented different types of protocols.

Date: 22-02-23

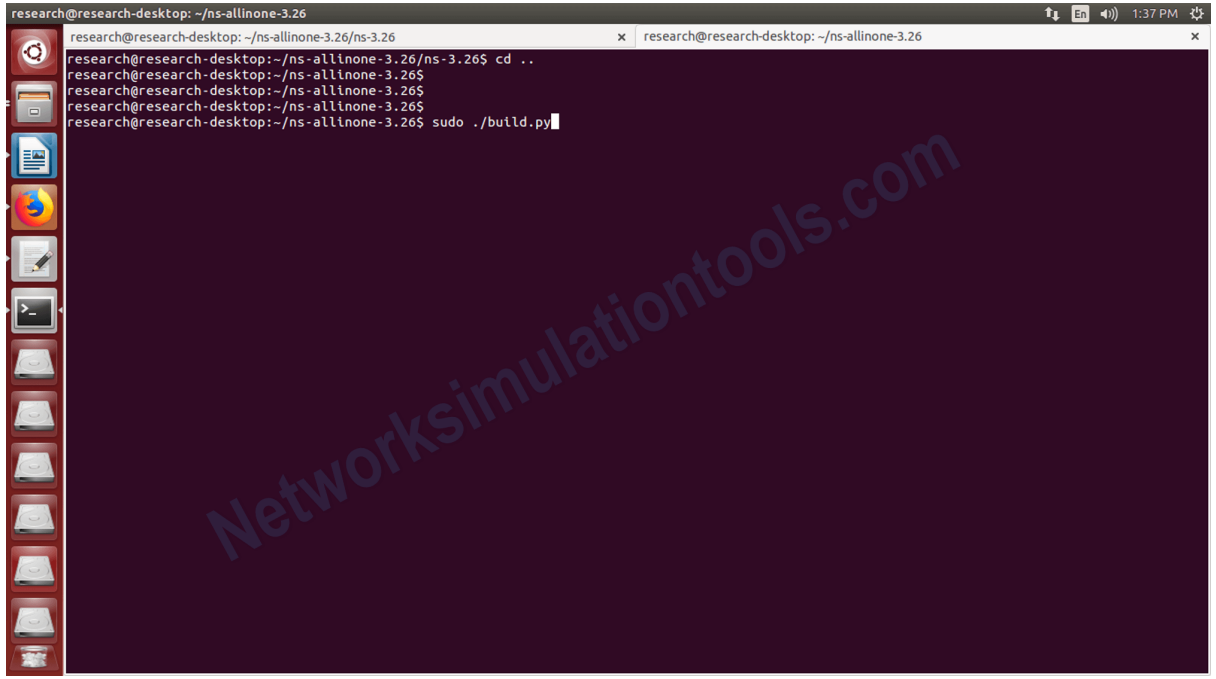
Client and Server Configurations using NS3

Using the NS3 ,we can generate various point to point codes such as first.cc ,second.cc and third.cc etc

The basic steps to be followed to generate the various client server networks are :

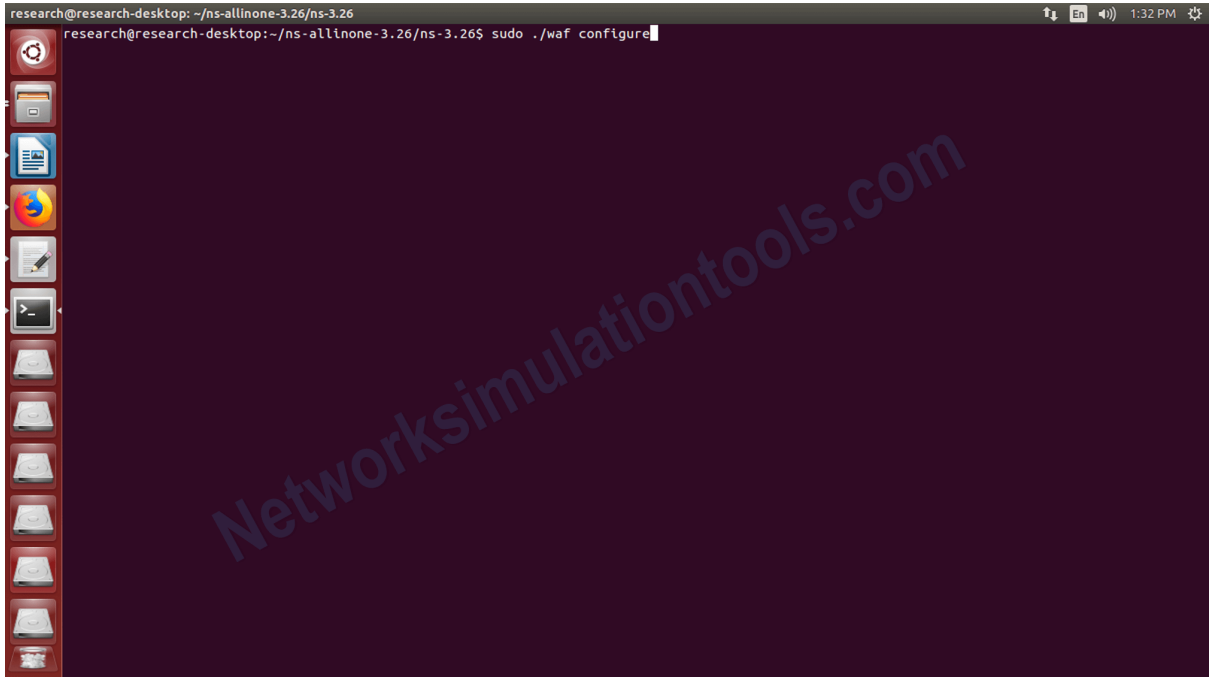
Install the NS-3.26

Initially , install the NS-3.26 tool by using the ns-allinone-3.26.tar.bz2 package. For make install use and execute the command ./b.py



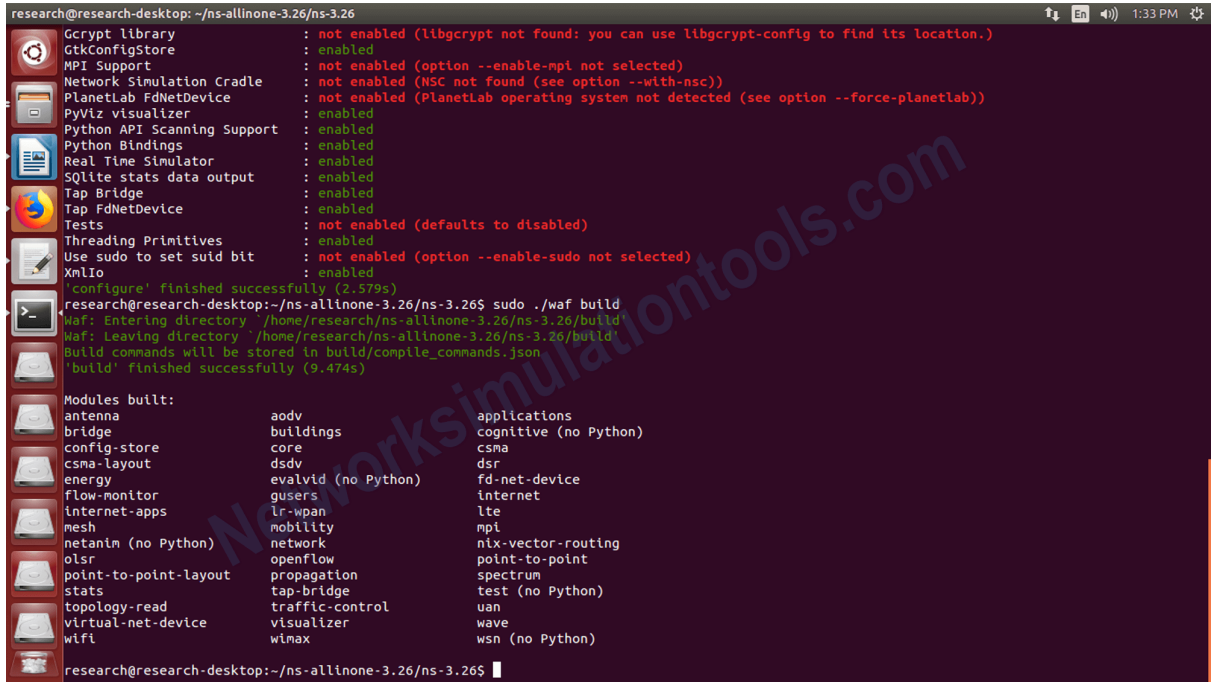
1. Configure the package

Change the ns-allinone installation location in the terminal, using the command cd ns-allinone-3.26/ns-3.26. And execute the command sudo ./waf configure to binding the python



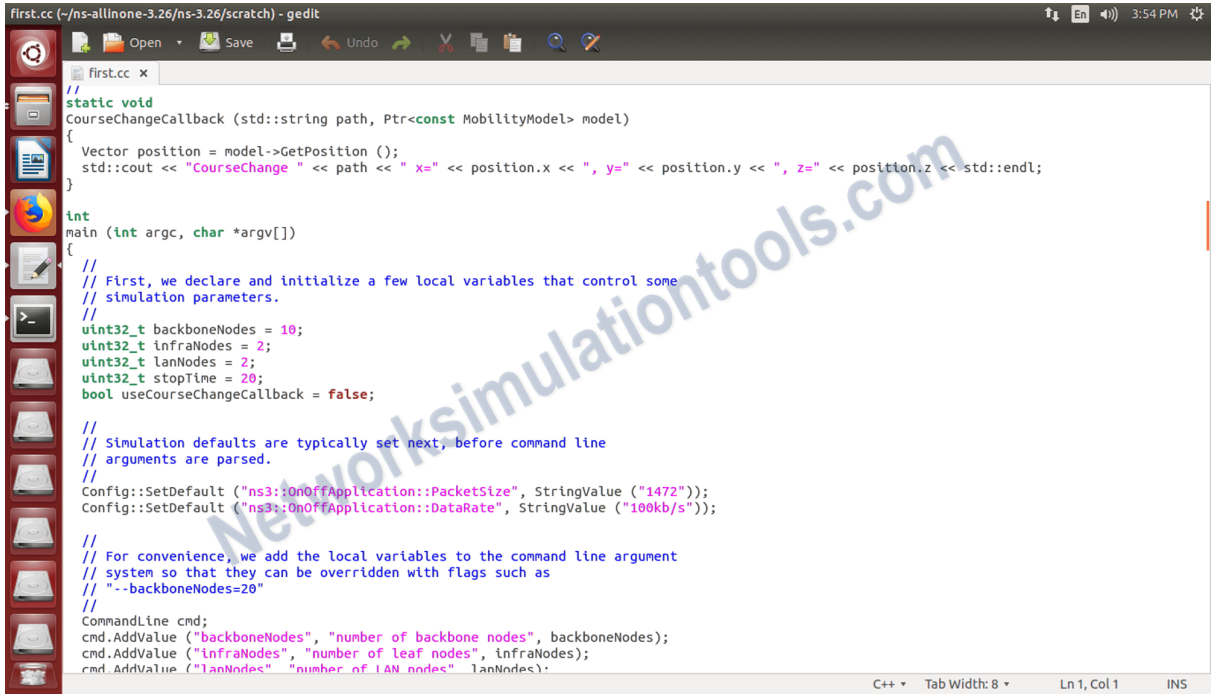
Build the package

For build the configured package by using the sudo ./waf build command in the ns-allinone-3.26/ns-3.26 . Package and view the build packages.



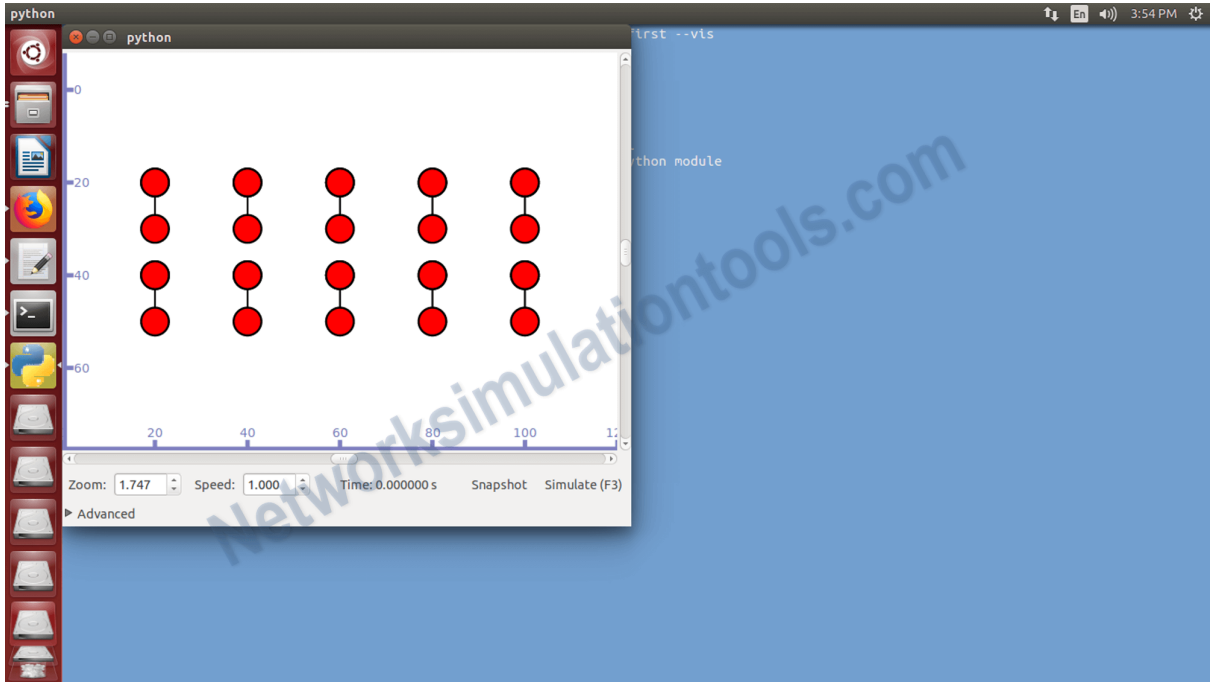
Create a main file

Next create new program file in the scratch folder. The program file stored with the file extension .cc. Include the header files for the needed modules.



1. Run the main command

Next, implement the foremost main file by using the command sudo ./waf –run first –vis to run the simulation result.



Execution of First.cc and its output

In the first.cc point to point echo client server connection is established n0----------n1 where n0 client and n1 server and data rate and delay can be configured .

```
FIRST.CC
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
```

```
//           10.1.1.0
// n0 -------------- n1
//    point-to-point
//
```

```
using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");

int
main (int argc, char *argv[])
{
  CommandLine cmd (__FILE__);
  cmd.Parse (argc, argv);

  Time::SetResolution (Time::NS);
  LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
  LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);

  NodeContainer nodes;
  nodes.Create (2);

  PointToPointHelper pointToPoint;
  pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
  pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

  NetDeviceContainer devices;
  devices = pointToPoint.Install (nodes);

  InternetStackHelper stack;
```

```
//           10.1.1.0
// n0 ------------------ n1
//    point-to-point
//
```

```
stack.Install (nodes);

  Ipv4AddressHelper address;
  address.SetBase ("10.1.1.0", "255.255.255.0");

  Ipv4InterfaceContainer interfaces = address.Assign (devices);

  UdpEchoServerHelper echoServer (9);

  ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
  serverApps.Start (Seconds (1.0));
  serverApps.Stop (Seconds (10.0));

  UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);
  echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
  echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
  echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

  ApplicationContainer clientApps = echoClient.Install (nodes.Get (0));
  clientApps.Start (Seconds (2.0));
  clientApps.Stop (Seconds (10.0));

  Simulator::Run ();
  Simulator::Destroy ();
  return 0;
}
```

```
vatsalya@Vatsalya:~/Desktop/ns-allinone-3.36.1/ns-3.36.1$ ./ns3 run scratch/first.cc
Consolidate compiler generated dependencies of target scratch_first
At time +2s client sent 1024 bytes to 10.1.1.2 port 9
At time +2.00369s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.00369s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.00737s client received 1024 bytes from 10.1.1.2 port 9
```

Execution of second.cc and its output

Second.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */


#include "ns3/core-module.h"                    // Default Network Topology
#include "ns3/network-module.h"                 //
#include "ns3/csma-module.h"                    //       10.1.1.0
#include "ns3/internet-module.h"                // n0 -------------- n1   n2   n3   n4
#include "ns3/point-to-point-module.h"          //    point-to-point  |    |    |    |
#include "ns3/applications-module.h"            //                    ================
#include "ns3/ipv4-global-routing-helper.h"     //                      LAN 10.1.2.0
using namespace ns3;


NS_LOG_COMPONENT_DEFINE ("SecondScriptExample");


int
main (int argc, char *argv[])
{
  bool verbose = true;
  uint32_t nCsma = 3;
```

```
CommandLine cmd (__FILE__);
  cmd.AddValue ("nCsma", "Number of \"extra\" CSMA nodes/devices", nCsma);
  cmd.AddValue ("verbose", "Tell echo applications to log if true", verbose);

  cmd.Parse (argc,argv);

  if (verbose)
    {
      LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
      LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);
    }
 Ipv4InterfaceContainer p2pInterfaces;
   p2pInterfaces = address.Assign (p2pDevices);

   address.SetBase ("10.1.2.0", "255.255.255.0");
   Ipv4InterfaceContainer csmaInterfaces;
   csmaInterfaces = address.Assign (csmaDevices);

   UdpEchoServerHelper echoServer (9);

   ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get (nCsma));
   serverApps.Start (Seconds (1.0));
   serverApps.Stop (Seconds (10.0));
```

```
UdpEchoClientHelper echoClient (csmaInterfaces.GetAddress (nCsma), 9);
  echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
  echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
  echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

  ApplicationContainer clientApps = echoClient.Install (p2pNodes.Get (0));
  clientApps.Start (Seconds (2.0));
  clientApps.Stop (Seconds (10.0));

  Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

  pointToPoint.EnablePcapAll ("second");
  csma.EnablePcap ("second", csmaDevices.Get (1), true);
```

```
nCsma = nCsma == 0 ? 1 : nCsma;


  NodeContainer p2pNodes;
  p2pNodes.Create (2);
  NodeContainer csmaNodes;
  csmaNodes.Add (p2pNodes.Get (1));
  csmaNodes.Create (nCsma);
  PointToPointHelper pointToPoint;
  pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
  pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
  NetDeviceContainer p2pDevices;
  p2pDevices = pointToPoint.Install (p2pNodes);
  CsmaHelper csma;
  csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
  csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));
  NetDeviceContainer csmaDevices;
  csmaDevices = csma.Install (csmaNodes);
  InternetStackHelper stack;
  stack.Install (p2pNodes.Get (0));
  stack.Install (csmaNodes);
  Ipv4AddressHelper address;
  address.SetBase ("10.1.1.0", "255.255.255.0");
  Simulator::Run ();
    Simulator::Destroy ();
    return 0;
  }
```

```
vatsalya@Vatsalya:~/Desktop/ns-allinone-3.36.1/ns-3.36.1$ ./ns3 run scratch/second.cc
Consolidate compiler generated dependencies of target scratch_second
At time +2s client sent 1024 bytes to 10.1.2.4 port 9
At time +2.0078s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.0078s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.01761s client received 1024 bytes from 10.1.2.4 port 9
```

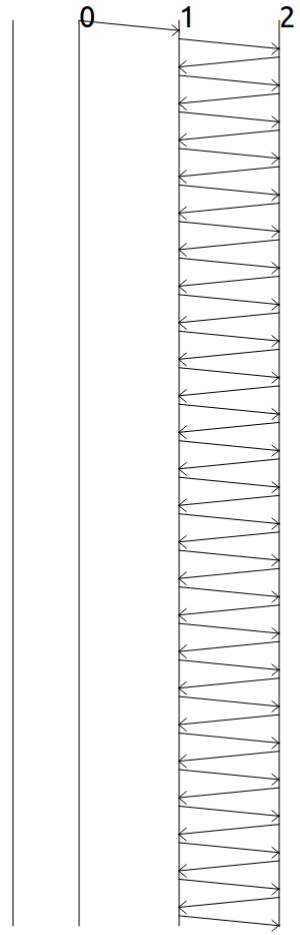
Intermediate Nodes N0-----N1-----N2

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/netanim-module.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");

int
main (int argc, char *argv[])
{
  CommandLine cmd;
  cmd.Parse (argc, argv);

  Time::SetResolution (Time::NS);
  LogComponentEnable ("UdpEchoClientApplication",
LOG_LEVEL_INFO);
  LogComponentEnable ("UdpEchoServerApplication",
LOG_LEVEL_INFO);

  NodeContainer nodes;
  nodes.Create (3);

  PointToPointHelper pointToPoint1;
  pointToPoint1.SetDeviceAttribute ("DataRate", StringValue
("5Mbps"));
  pointToPoint1.SetChannelAttribute ("Delay", StringValue
("2ms"));
```

```
  NetDeviceContainer devices1;
  devices1 = pointToPoint1.Install (nodes.Get(0),nodes.Get(1));

 NetDeviceContainer devices2;
  devices2 = pointToPoint2.Install (nodes.Get(1),nodes.Get(2));

  InternetStackHelper stack;
  stack.Install (nodes);

  Ipv4AddressHelper address1;
  address1.SetBase ("10.1.1.0", "255.255.255.0");

 Ipv4AddressHelper address2;
  address2.SetBase ("10.1.2.0", "255.255.255.0");


  Ipv4InterfaceContainer interfaces1 = address1.Assign
(devices1);
 Ipv4InterfaceContainer interfaces2 = address1.Assign
(devices2);

  UdpEchoServerHelper echoServer (9);

  ApplicationContainer serverApps = echoServer.Install
(nodes.Get (2));
  serverApps.Start (Seconds (1.0));
  serverApps.Stop (Seconds (10.0));
```

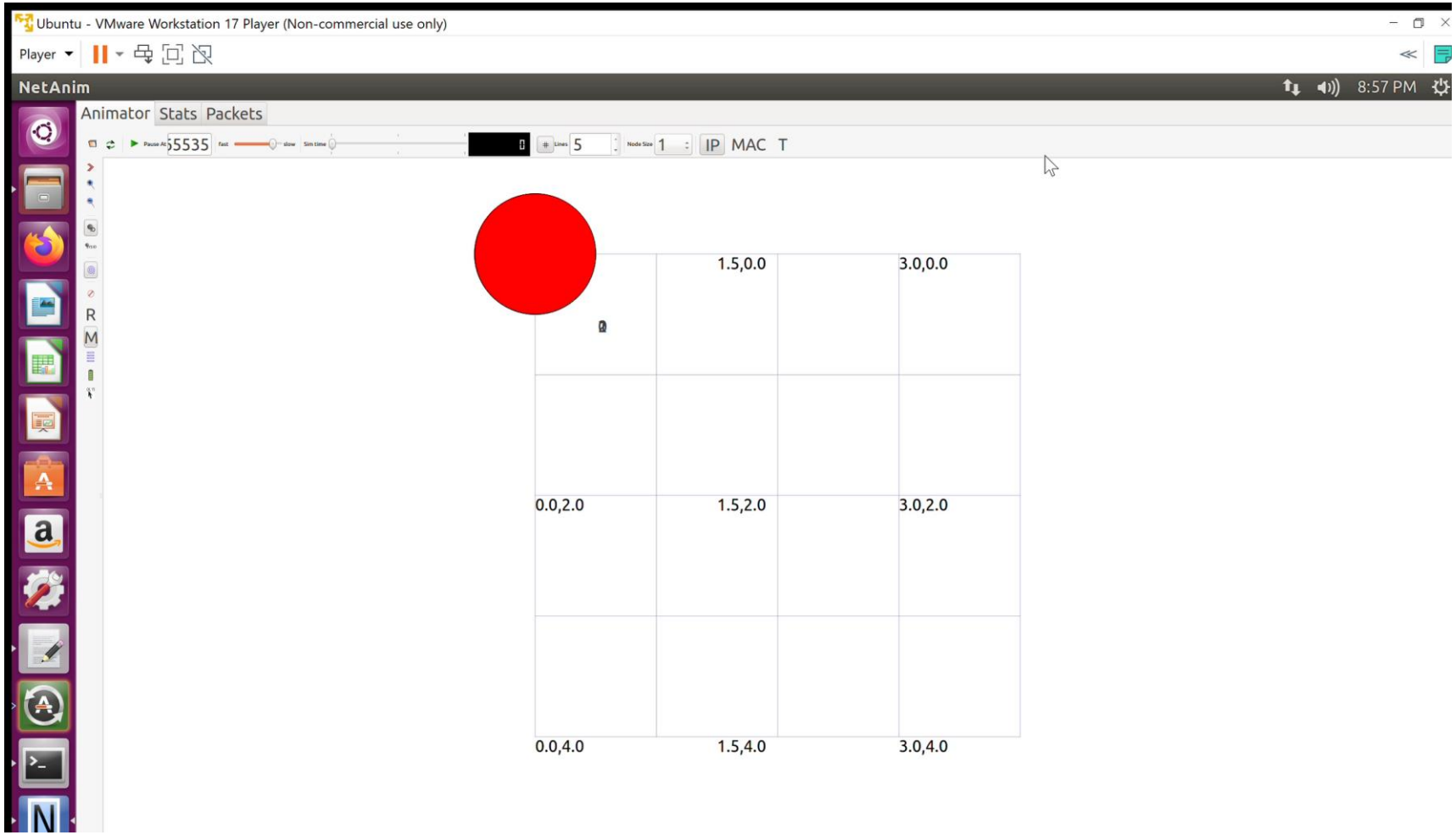
System Settings

```
  UdpEchoClientHelper echoClient (interfaces2.GetAddress (1),
9);
  echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
  echoClient.SetAttribute ("Interval", TimeValue (Seconds
(1.0)));
  echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

  ApplicationContainer clientApps = echoClient.Install
(nodes.Get (0));
  clientApps.Start (Seconds (2.0));
  clientApps.Stop (Seconds (10.0));

  AnimationInterface anim ("anim1.xml");
  anim.SetConstantPosition(nodes.Get(0),1.0,2.0);
  anim.SetConstantPosition(nodes.Get(1),2.0,3.0);
 anim.SetConstantPosition(nodes.Get(2),3.0,4.0);


  Simulator::Run ();
  Simulator::Destroy ();
  return 0;
}
```



port Tab

	From Id	To Id
1	0	1
2	1	2
3	2	1
4	1	2
5	2	1
6	1	2
7	2	1
8	1	2
9	2	1
10	1	2
11	2	1
12	1	2
13	2	1
14	1	2
15	2	1
16	1	2
17	2	1
18	1	2
19	2	1
20	1	2
21	2	1
22	1	2
23	2	1
24	1	2
25	2	1
26	1	2
27	2	1


Ubuntu - VMware Workstation 17 Player (Non-commercial use only)
Player
NetAnim
8:57 PM
Animator
Stats
Packets
IP
MAC
T
1.5,0.0
3.0,0.0
0.0,2.0
1.5,2.0
3.0,2.0
0.0,4.0
1.5,4.0
3.0,4.0

Work-let Name: Performance evaluation of DSCP values in various network

Worklet Details

1. Worklet ID: CP214MS
2. College Name: RAMAIAH INSTITUTE OF TECHOLOGY

KPIs achieved till now

Build an android application to setup the DSCP values

Any Challenges/ Issues faced

Facing slight issues with the API's

Next Steps

To finalize the build of the application and analize it over different service providers

Key Achievements/ Outcome till now

- Learned how to use android studio
- Created server using Java
- Able to setup the client server application which is also build on Java

Date: 20/06/23

Application setup in Android Studio

How it works

- Interface asks for the DSCP values which can be manually entered by the user
- Then there is a send option to send the DSCP values entered by the user to the server
- In the backend we have implemented auto generation of a 2mb file which is delivered according to the DSCP values entered by the user

Client side

```
package com.example.prims_june;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;

import android.os.AsyncTask;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

import androidx.appcompat.app.AppCompatActivity;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;

public class MainActivity extends AppCompatActivity {

    private EditText dscpEditText;
    private Button sendButton;
    private TextView responseTextView;
```

```
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        dscpEditText = findViewById(R.id.dscpEditText);
        sendButton = findViewById(R.id.sendButton);
        responseTextView = findViewById(R.id.responseTextView);

        sendButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                String dscpValue = dscpEditText.getText().toString();
                int dscp = Integer.parseInt(dscpValue);

                SendDataTask task = new SendDataTask(dscp);
                task.execute();
            }
        });
    }

    private class SendDataTask extends AsyncTask<Void, Void, String> {
        private int dscp;

        public SendDataTask(int dscp) {
            this.dscp = dscp;
        }
```

```
@Override
    protected String doInBackground(Void... voids) {
        Socket socket = null;
        try {
            socket = new Socket("192.168.244.1", 12345);

            // Set DSCP value for the socket
            socket.setTrafficClass(dscp);

            // Send the data packet
            DataOutputStream outputStream = new DataOutputStream(socket.getOutputStream());
            byte[] dataPacket = generateDataPacket();
            outputStream.write(dataPacket);
            outputStream.flush();

            // Receive the response from the server
            DataInputStream inputStream = new DataInputStream(socket.getInputStream());
            String response = inputStream.readUTF();

            inputStream.close();
            outputStream.close();

            return response;
        }
```

```
        catch (IOException e) {
                e.printStackTrace();
                return "Error: " + e.getMessage();
            } finally {
                if (socket != null) {
                    try {
                        socket.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
    @Override
        protected void onPostExecute(String response) {
            responseTextView.setText(response);
        }

        private byte[] generateDataPacket() {
            // Generate a 2MB data packet
            // Replace this with your own logic to generate the data packet
```

```
            byte[] dataPacket = new byte[2 * 1024 * 1024];
            // Fill the data packet with your data

            return dataPacket;
        }
    }
}
```

Output for the Server site