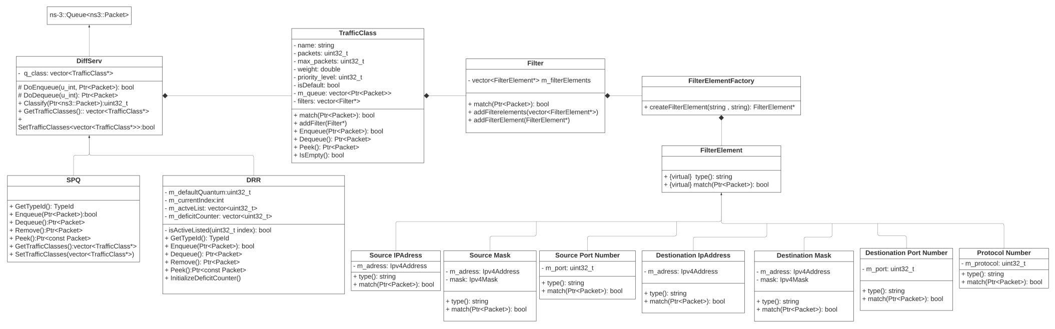
CS621 Spring-2023 Project2 Final Report

Team Members: Deep Mistry, Gizem Hazal Senturk

DESIGN

The ns3 classes we have implemented/used are:
- Queue: This is the parent class for all the queues we have implemented. DiffServ class inherited the ns3 Queue class. Since this class will have the QoS algorithm's classes as subclasses, we implemented Schedule and Classify methods in DiffServ class and left the implementation of the public Enqueue() and Dequeue() methods to the subclasses SPQ and DRR since they will implement them differently. The protected DoEnqueue and DoDequeue methods are implemented in DiffServ facilitating the enqueueing and dequeuing of the packets from the TrafficClass's queue.
- Packet: Since the QueueMode is removed from the recent release of ns3, our program solely relies on packets. We used this class of ns3 as our main elements of the queues. These packets will be the main argument for the filter element match methods. They will be checked against certain values in the packet header vs the values in the filter element.
- Point-to-point module: We used this module to create the point-to-point links between the nodes. We used the NetDeviceContainer to create the links between the nodes.
- Log Component: We used this component to log the errors and warnings. We used the NS_LOG_COMPONENT_DEFINE macro to define the log component.
- Ipv4 Address: NS3 IPV4 class is the class that we used to assign the IP addresses to the nodes. We used the Ipv4AddressHelper class to assign the IP addresses and masks to the nodes.
- UDP header: We used this header to create the UDP packets.
- TCP header: We used this header to create the TCP packets.
- PPP(Point to point protocol) header: Used to create point-to-point links.
- Ipv4 header: Used to create the Ipv4 packets.

The UML Diagram of our classes are as below:


ns-3::Queue<ns3::Packet>
DiffServ
- q_class: vector<TrafficClass*>
# DoEnqueue(u_int, Ptr<Packet>): bool
# DoDequeue(u_int): Ptr<Packet>
+ Classify(Ptr<ns3::Packet>):uint32_t
+ GetTrafficClasses():: vector<TrafficClass*>
+ SetTrafficClasses<vector<TrafficClass*>>:bool
TrafficClass
- name: string
- packets: uint32_t
- max_packets: uint32_t
- weight: double
- priority_level: uint32_t
- isDefault: bool
- m_queue: vector<Ptr<Packet>>
- filters: vector<Filter*>
+ match(Ptr<Packet>): bool
+ addFilter(Filter*)
+ Enqueue(Ptr<Packet>): bool
+ Dequeue(): Ptr<Packet>
+ Peek(): Ptr<Packet>
+ IsEmpty(): bool
Filter
- vector<FilterElement*> m_filterElements
+ match(Ptr<Packet>): bool
+ addFilterelements(vector<FilterElement*>)
+ addFilterElement(FilterElement*)
FilterElementFactory
+ createFilterElement(string , string): FilterElement*
FilterElement
+ {virtual} type(): string
+ {virtual} match(Ptr<Packet>): bool
SPQ
+ GetTypeId(): TypeId
+ Enqueue(Ptr<Packet>):bool
+ Dequeue():Ptr<Packet>
+ Remove():Ptr<Packet>
+ Peek():Ptr<const Packet>
+ GetTrafficClasses():vector<TrafficClass*>
+ SetTrafficClasses(vector<TrafficClass*>)
DRR
- m_defaultQuantum:uint32_t
- m_currentIndex:int
- m_actveList: vector<uint32_t>
- m_deficitCounter: vector<uint32_t>
- isActiveListed(uint32_t index): bool
+ GetTypeId(): TypeId
+ Enqueue(Ptr<Packet>): bool
+ Dequeue(): Ptr<Packet>
+ Remove(): Ptr<Packet>
+ Peek():Ptr<const Packet>
+ InitializeDeficitCounter()
Source IPAdress
- m_adress: Ipv4Address
+ type(): string
+ match(Ptr<Packet>): bool
Source Mask
- m_adress: Ipv4Address
- mask: Ipv4Mask
+ type(): string
+ match(Ptr<Packet>): bool
Source Port Number
- m_port: uint32_t
+ type(): string
+ match(Ptr<Packet>): bool
Destionation IpAddress
- m_adress: Ipv4Address
+ type(): string
+ match(Ptr<Packet>): bool
Destination Mask
- m_adress: Ipv4Address
- mask: Ipv4Mask
+ type(): string
+ match(Ptr<Packet>): bool
Destionation Port Number
- m_port: uint32_t
+ type(): string
+ match(Ptr<Packet>): bool
Protocol Number
- m_protocol: uint32_t
+ type(): string
+ match(Ptr<Packet>): bool

DESIGN ISSUES, CHALLENGES & LIMITATIONS

The most challenging part of the project was the drastic changes with the previous version of ns3 and the recent release which was on March 2023. The classes, their public, virtual, protected and private methods changed a lot. We had to change our design and implementation according to the new version. Also, the documentation of the new version being not descriptive enough and since that is a recent release there were not many examples to look at to understand the new version. The main challenge in the project was the steep learning curve of the new version of ns3.

TO IMPROVE DIFFSERV

We can implement more QoS algorithms. Currently, all QoS services are using the same classifier. We can make DiffServ an interface without any method implementation. In this way, it can act as a blueprint and developers can write their own classifiers, doenqueue do dequeue methods and this makes the software pluggable and more modular. Also, keeps the code open for extensions without changing the existing code.

IMPLEMENTATION DETAILS

- Config file: The config file is used mainly to get the network configurations to be installed to the router node later on. It is an XML file. The structure of the config file is as follows:
-

The parameters can be introduces in the config file are the following and it is important to keep the naming convention the same:
Source IP Address:
`<source_ip_address> value of source_ip_address </source_ip_address>`
Destination IP Address:
`<destination_ip_address> value of destination_ip_address </destination_ip_address>`
Source Port:
`<source_port> value of source_port </source_port>`
Destination Port:
`<destination_port> value of destination_port </destination_port>`
Protocol: the value of protocol
Source Mask:
`<source_mask> value of source_mask </source_mask>`
Destination Mask:
`<destination_mask> value of destination_mask </destination_mask>`

An example config file is as below:

```
<queues>
  <queue>
    <name>Queue1</name>
    <priority>3</priority>
    <filters>
      <filter_elements>
        <destination_port>2</destination_port>
      </filter_elements>
      <filter_elements>
        <source_ip_address>1.0.9.3</source_ip_address>
        <source_port>8080</source_port>
      </filter_elements>
      <filter_elements>
        <source_ip_address>1.0.19.3</source_ip_address>
        <source_port>8080</source_port>
      </filter_elements>
    </filters>
  </queue>
  <queue>
    <name>Queue2</name>
    <priority>2</priority>
    <filters>
      <filter_elements>
        <destination_port>1</destination_port>
      </filter_elements>
      <filter_elements>
        <source_ip_address>1.0.9.55</source_ip_address>
        <source_port>8080</source_port>
      </filter_elements>
    </filters>
  </queue>
</queues>
...
```

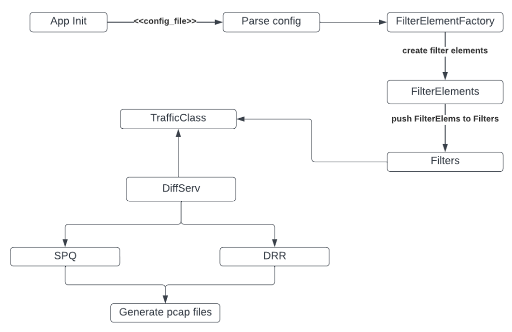
The program requires a config file to be given as a command line argument. Program flow considering the implementation of the classes and their functionalities:

Following that in the main function, we parse the config file, and the following steps are processed:
- Create the empty TrafficClass vector,
- Create a vector of Filters
- Create the FilterElements using the FilterElementFactory, and add them to the related filter and then Filter is added to the Filter vector
- Add the Filter vector to the TrafficClass, and then TrafficClass is added to the TrafficClass vector
- Until all the Queues are created from the config file repeat this process
- Create the QoS algorithm object based on the config file - if priority is defined in the config the SPQ object will be created as a subclass of DiffServ
If the weight is defined in the config file, the DRR object will be created as a subclass of DiffServ.
- Then start creating the topology and different scenarios that are explained in the project specification.

FilterElementFactory: We introduced a class, to the classes expected in the UML diagram, in order to create the filter elements with a factory design pattern.
The factory returns a FilterElement class depending on the parameters passed to the factory.

SPQ: SPQ is a subclass of DiffServ. It is a strict priority queue. Uses the Classify, DoEnqueue, DoDeqeueue methods from DiffServ.
And further implements Enqueue and Dequeue methods.

DRR: DRR is a subclass of DiffServ. It is a deficit round-robin queue. Uses the Classify, DoEnqueue, DoDeqeueue methods from DiffServ.



And further implements Enqueue and Dequeue methods. The logic is implemented based on the paper Efficient Fair Queuing Using Deficit Round Robin, Shreedhar et al.(1995).

isDefault: This parameter allows the packets that do not match with any filters to be assigned to a default queue.
We preferred to pass this parameter to be defined in

the config file as a queue property.

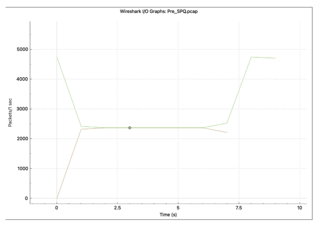
The classes not mentioned here loosely follow the UML diagram. The program processes are shown in the above diagram.

SIMULATION VALIDATION AND VERIFICATION

Throughout the development process, we used the NS3 logger to print out the errors and warnings. We also used the NS3 Flow Monitor to monitor the traffic and the queues. Since the output of the program is the pcap files that recorded traffic, we used Wireshark to analyze the traffic and the queues. And in order to verify the flows, we plotted the I/O graphs of the flows. For the SPQ, the plots were as expected. We saw the strict prioritization of the flow defined in the configuration file- having two queues defined with one being prior to the other. The moment we started the application that is supposed to have the highest priority, the other flows linearly decreased and stopped until the application with the highest priority finished. And the other flows started again.

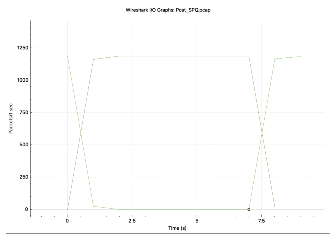
Considering the two figures below and the parameters set for the topology, we see that in pre spa they are given the same bandwidth and once the packets sent from the router we can see the QoS being implemented. Higher priority queue among too, considering the 1 second delay, starves the lower priority queue upon application start. Then once it sends all the packets, lower priority queue starts to take over.

Pre SPQ



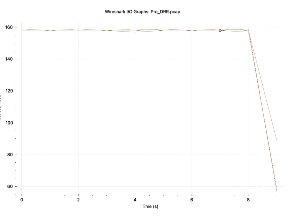
Topology parameters:
Data rate (Router- Server) : 10 Mbps
Data rate (Client-Router): 40 Mbps
Packet size: 1024
Packet frequency: 10000/sec

Post SPQ



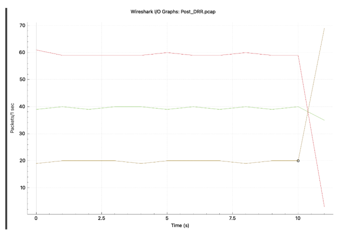
For the DRR, we introduced 3 queues in the scenario. Queues are served with quanta values proportional to their weights. Weights are defined in the configuration file. We followed the same process for validation as it is in the SPQ scenario. And again we validated the results with the plots. The results were as expected.
The queue with the highest weight was served with higher bandwidth than the other queues. And the queue with the lowest weight was served with the lowest bandwidth.
For DRR we can see in the pre-plot that all the flows started at the same time and there were no differences in the client-router link priorities. But once the QoS activate on the link between router and server we can see the bandwidth is allocated according to the ratio of weights defined in the configuration file.

Pre DRR



Topology parameters:
Data rate (Router- Server) : 1 Mbps
Data rate (Client-Router): 4 Mbps
Packet size: 1024
Packet frequency: 10000/sec

Post DRR



At the end of the project, we removed the output statements and Flow Monitor. So the only output that will be saved to the directory of the ns3 would be pcap files.

In both scenarios, if a isDefault value is set to true, we destined the packets with no matches with the filters to that queue to be served. This can be defined by the user in the config file. Knowing the common practice is to make the lowest priority queue the default queue, we let the user define the default queue in the config file. if the value is not defined or defined as false, the packets with no matches with the filters are dropped.

API USAGE

In order to use the classes we implemented, the user needs to add the required ns3 header files to the main.cc which is nwp2.cc in our case.
They also need to add the header file for the DiffServ, TrafficClass, Filter, SPQ, and DRR classes. The parser that we used was RapidXML,
and again in the main file, the header classes are added. The queue configurations mentioned in the config file are used to create the filter elements. So defining a proper config file with the matching types
is important since we use the factory design pattern. The factory creates those elements by types. If the user wants to create an additional filter element,
they just need to write the Filter Element class and register that class to the factory with the type.
In order to use the QoS algorithm classes that we implemented, they create an object of the QoS algorithm class and set the TrafficClass by calling the setTrafficClasses method and parsing the vector as a parameter.
Then they can use the scenarios defined in the nwp2.cc file. They need to pass the object to the simulation method.

ALTERNATIVE DESIGN

We can use the DiffServ class solely as an interface. In this way, any developer can implement their own QoS algorithms and use the DiffServ class as an interface.
This would make the program pluggable and more modular. Also, we can make the TrafficClass class as an interface and make the QoS algorithms to inherit from that class.

REFERENCES

Efficient Fair Queuing Using Deficit Round Robin, Shreedhar,M., Varghese, G., 1995
Simulating Strict Priority Queueing, Weighted Round Robin and Weighted Fair Queueing with NS-3, Chang,R., Pournaghshband,V., 2017
NS3 Documentation, https://www.nsnam.org/docs/release/3.38/doxygen/index.html