CS621 Network Programming - A Base Class to Simulate Differentiated Services

Alberto Delgado Jason McGowan

May 2023

1 Introduction

The aim of this project is to enhance our understanding and practical skills in working with a network simulator, specifically in implementing differentiated services. The project involves the creation of a base class for differentiated services in ns-3, followed by the implementation of two Quality of Service (QoS) mechanisms: Strict Priority Queuing (SPQ) and Deficit Round Robin (DRR).

The base class, named DiffServ, provides the foundational functionalities required to simulate differentiated services, which include traffic classification and scheduling. Upon the successful implementation of the base class, we will utilize it to implement SPQ and DRR.

The output goals of the project are twofold. Firstly, we will demonstrate the correct functioning of the SPQ and DRR mechanisms by validating and verifying them through ns-3 simulation runs. This will involve creating and running various simulation scenarios, which may be automated through scripting.

Secondly, we aim to demonstrate our understanding of the ns-3::Queue API reference and the closely related ns-3::DropTailQueue API reference, which form the basis of our DiffServ class.

Ultimately, the project is designed to provide a hands-on experience of implementing and testing differentiated services in a network simulator, thus equipping us with the practical skills needed to work effectively with QoS mechanisms in real-world networking environments.

2 Design

The aim of our project was to create a DiffServ implementation in ns-3 that would be agnostic to other modules, meaning that our work would not create new dependencies between the existing modules. This is essential for maintaining a clean, modular, and maintainable codebase.

The centerpiece of our design is the *DiffServ* class, which serves as the base class for our project. This class is designed to inherit from the existing *Queue* class, allowing all Quality of Service (QoS) queues to inherit from DiffServ. This

design decision is based on the need for all queues to conform to the same queue abstraction.

Another key aspect of our design is the use of *Traffic Classes*. These are essentially specialized queues that can handle different types of network traffic according to their specific requirements. Each TrafficClass holds a list of filters, which are used to classify incoming packets and decide which queue they belong to.

We've also designed a *Filter* and *FilterElement* mechanism. Filters are composed of multiple FilterElements, which each specify a condition that a packet must meet. If a packet meets all the conditions specified by a Filter's FilterElements, it is considered a match for that filter.

Our design includes an *Examples* folder which contains examples of SPQ and DRR simulations. These examples serve as practical demonstrations of how our DiffServ implementation can be used in real-world scenarios.

Furthermore, we've also included a *Tests* folder that contains both unit tests and end-to-end tests. These tests are essential for validating the correct operation of our implementation and ensuring that it meets the specified requirements.

3 Design Issues/Challenges/Limitations

Our implementation, despite its strengths, has several known issues and limitations that could be addressed given additional time.

The first limitation lies in the DiffServ and TrafficClass design. Currently, the classes do not implement a mechanism for dynamic reconfiguration, meaning that once the traffic classes and their corresponding parameters are set from the configuration file, they cannot be changed during runtime. A more flexible design would allow dynamic reconfiguration of traffic classes and their parameters, providing the ability to adjust to changing network conditions or QoS requirements.

Secondly, in the DiffServ class, the 'DoRemove' and 'DoPeek' methods are not implemented. While this does not affect the basic functionality of the queue management, it does limit the control and visibility a user has over the queued packets. Implementing these methods would allow users to remove specific packets from the queue or peek at the next packet to be dequeued without actually dequeuing it.

Additionally, the current Filter implementation only supports a limited set of filter types, specifically protocol, source and destination IP masks, and source and destination ports. In future iterations, we would extend the FilterElement class to support a wider variety of filter types, such as Time to Live (TTL), Type of Service (ToS), or other IP and TCP/UDP header fields. This would make our DiffServ implementation more versatile in handling diverse QoS requirements.

Finally, there is a potential memory management issue due to the use of raw pointers for TrafficClass objects and FilterElements. This could lead to memory leaks if these objects are not properly deallocated. Moving forward, we would

consider replacing raw pointers with smart pointers to improve memory safety and simplify memory management.

These challenges represent opportunities for future enhancement and refinement of our implementation. Despite these current limitations, our implementation provides a robust foundation for QoS queue management in ns-3.

4 Suggestions to Improve DiffServ

Below are some proposed enhancements for the DiffServ module:

1. **Dynamic Reconfiguration:** The existing static configuration of traffic classes and parameters can be limiting, especially in networks with highly variable traffic conditions. To accommodate such scenarios, a mechanism to allow dynamic reconfiguration of traffic classes and their attributes could be introduced. This would allow administrators to adjust traffic management policies on-the-fly to respond to changing network conditions. This could include the ability to add or remove traffic classes, change priority levels, and adjust maximum packet and byte counts, among other parameters.

2. **Enhanced Queue Management:** Currently, the DiffServ module only supports a basic queue management scheme. Implementing more advanced queue management algorithms, such as Random Early Detection (RED) or Explicit Congestion Notification (ECN), could help improve network performance, especially under heavy load.

3. **Traffic Shaping:** In addition to prioritizing different classes of traffic, DiffServ could implement traffic shaping mechanisms to smooth out bursty traffic and reduce the chances of network congestion.

4. **Performance Metrics:** The DiffServ module could be enhanced to provide more detailed metrics and statistics about traffic classes, such as queue lengths, packet drop rates, and latency. This data could be useful for network administrators for troubleshooting and capacity planning.

5. **Improved Filter Matching:** The filter matching mechanism could be enhanced to support more complex matching criteria. This could include the ability to match regular expressions or wildcards.

5 Implementation Details

5.1 DiffServ

The core of our DiffServ implementation is the *DiffServ* class, which inherits from the existing ns-3 *Queue* class. The DiffServ class introduces a list of TrafficClasses, each of which contains a set of filters for classifying incoming packets.

Each TrafficClass maintains its own queue of packets and includes methods for enqueueing and dequeueing packets. When a packet is enqueued, it is checked against the TrafficClass's filters to determine if it matches the criteria for that class.

The DiffServ class also includes a *Classify* method which evaluates incoming packets and determines which TrafficClass they belong to. This is done by checking the packet against each TrafficClass's filters in turn. If a packet matches the filters for a particular TrafficClass, it is assigned to that class.

Our implementation includes a *Schedule* method which is left abstract in the DiffServ class and is expected to be implemented by each specific QoS queue. This method determines the logic for which packet should be dequeued next.

For configuring the DiffServ module and its associated TrafficClasses and filters, we have employed a JSON-based configuration file. This configuration file allows the user to specify the parameters for each TrafficClass, such as its priority level, weight, maximum packet and byte counts, and filters. Our implementation uses the nlohmann::json library for parsing the configuration file.

We have included a variety of FilterElements for classifying packets, including destination IP mask, destination port, protocol, source IP mask, and source port. These FilterElements can be easily extended to support additional types of filtering as needed.

5.2 Strict Priority Queuing (SPQ)

In the SPQ implementation, traffic classes are stored in a vector (q_class). They are sorted in descending order according to their priority levels during initialization. The *Schedule* function iterates over the priority levels from highest to lowest. For each priority level, it iterates over the traffic classes. If the TrafficClass matches the current priority level and has packets in its queue, it dequeues and returns the packet. If no packets match the highest priority level, it moves on to the next level, and so on. If none of the traffic classes have any packets, it returns null.

The SPQ class also includes methods for enqueuing, dequeuing, and removing packets, as well as a method for peeking at the front of the queue.

5.3 Deficit Round Robin (DRR)

The DRR implementation maintains a list of active traffic classes. When a packet is enqueued, the *Enqueue* function first classifies the packet to find its corresponding traffic class. It then tries to enqueue the packet. If the enqueueing is successful, it checks if the TrafficClass is already in the active list. If not, it adds it to the active list and sets its deficit counter to zero.

The *Dequeue* function works in a round-robin fashion, iterating over the active list of traffic classes. For each traffic class, it adds the weight of the TrafficClass to its deficit counter. If the number of bytes in the traffic class's queue is greater than its deficit counter, it moves on to the next traffic class.

If the number of bytes is less than or equal to the deficit counter, it dequeues a packet, subtracts the size of the packet from the deficit counter, and returns the packet. If the traffic class's queue is now empty, it resets the deficit counter and removes the TrafficClass from the active list.

The Dequeue function works in a round-robin fashion, iterating over the active list of traffic classes. When entering a new TrafficClass, it adds the weight of the TrafficClass to its deficit counter. If the number of bytes in the TrafficClass's head packet is greater than its deficit counter, it moves on to the next traffic class. If the number of bytes is less than or equal to the deficit counter, it dequeues a packet, subtracts the size of the packet from the deficit counter, and returns the packet. Further Dequeue operations will remain in the current queue until a packet exceeds its deficit counter or is emptied. If the traffic class's queue is now empty, it resets the deficit counter and removes the TrafficClass from the active list.

The DRR class also includes placeholder methods for removing packets and peeking at the front of the queue, which are not implemented in the provided code.

6 Simulation Validation and Verification

The validation and verification of the Strict Priority Queueing (SPQ) and Deficit Round Robin (DRR) implementations were conducted using ns3 simulations. The core simulation setup for both involved a network with three nodes.

For the SPQ validation, the network was designed to reflect a traffic scenario where there would be a distinct prioritization of packets. Two of the nodes were configured as UDP traffic sources while the third node, situated between the two source nodes, acted as the forwarding agent. This node was configured with the SPQ queuing discipline, reading configuration from a JSON file that defined the traffic classes, their priority levels, and packet filters. The simulation was designed to terminate if no traffic classes were specified in the configuration file.

The traffic sources were programmed to generate high-priority and low-priority traffic at different intervals. Despite the high-priority traffic starting later, the SPQ mechanism should ensure its packets get forwarded ahead of the low-priority packets due to their higher priority level.

The DRR validation followed a similar network setup. The difference here was the queuing discipline used at the forwarding node, which in this case was DRR. The configuration file for DRR would specify weights for the traffic classes instead of priority levels.

The simulations generated packet capture (pcap) files that were then analyzed using Wireshark. This analysis verified that the SPQ and DRR mechanisms were correctly prioritizing and scheduling the packet transmissions as per their design. Furthermore, these pcap files were used to plot graphs, providing a visual representation and further evidence of the correct functioning of the SPQ and DRR implementations.

7 API Usage

To utilize the SPQ and DRR implementations in ns3, the user must be familiar with the key components of the API. Here are the primary elements:

- **SPQ and DRR classes**: These classes extend the base 'Queue' class in ns3. An instance of these classes can be set as the queuing discipline for a given net device.
- **Configuration file**: This JSON file is necessary to set up traffic classes for SPQ or DRR. For SPQ, each TrafficClass includes information about priority levels and packet filters, while for DRR, the weights for each TrafficClass are specified.
- **Enqueue and Dequeue methods**: These methods are overridden in the SPQ and DRR classes and handle the addition and removal of packets from the queue. They employ the respective scheduling algorithm (SPQ or DRR) when dequeuing packets.
- **Classify method**: This method is responsible for determining the TrafficClass of an incoming packet. It is used in the Enqueue method to determine which queue the packet should be placed in.

To use these classes, a user would generally create an instance of either SPQ or DRR, read in the appropriate configuration file to set up traffic classes, and then set this instance as the queue discipline for a particular net device. Then, when packets are sent over this device, they will be scheduled according to either the SPQ or DRR algorithm, based on the TrafficClass they belong to.

8 Alternative Design

For alternative designs and improvements, please see Section 4.

9 References

The development of this project was heavily influenced by a range of resources, including software libraries, documentation, and academic papers. Here are some key resources that were invaluable during the development process:

- **NS-3 Library**: Our implementation of SPQ and DRR scheduling algorithms in the DiffServ model was developed using the NS-3 network simulator. The library provided the fundamental building blocks to model network behavior accurately.
- **NS-3 Documentation, Tutorials, and Examples**: These resources provided by the NS-3 project were critical in understanding how to use the library effectively. They offered practical examples and explanations of

the network simulator's core concepts, which were instrumental in shaping our design and implementation approach.

- **Differentiated Service Queuing Disciplines in NS-3**: This white paper by Robert Chang, Mahdi Rahimi, and Vahab Pournaghshband provided in-depth insights into the design and functionality of Differentiated Service Queuing Disciplines. The paper's analysis and examples provided a solid foundation for our implementation of the DiffServ model in NS-3.