DiffServ: A Base Class in NS-3

Ashley Radford
University of San Francisco

ABSTRACT

Strict priority queuing (SPQ) and deficit round robin (DRR) are two differentiated services that provide quality of service for specific selections of traffic. This paper describes the designs and implementations of these two mechanisms in ns-3, a discrete event network simulator that provides a simulation environment for modeling and analyzing network systems. Since there is a large amount of overlap between these types of services, we first design a base class in which SPQ and DRR (and additional differentiated services) can utilize, thus reducing duplicated structure and logic.

1 INTRODUCTION

DiffServ is a base class that provides the core functionalities required to simulate differentiated services in ns-3. It is a subclass of ns-3's abstract base class Queue [2]. We will first describe its design, the additional public (and protected) class methods added, and discuss improvements. Next, we will discuss the design, implementation, and simulation results of DiffServ's subclasses SPQ and DRR.

2 DIFFSERV DESIGN

DiffServ is an abstract class that contains a vector called q_class. This vector is comprised of TrafficClass pointers. TrafficClass is an additional class that represents a packet queue. It contains metadata such as the number of packets currently in its queue, the maximum number of packets that it can hold, its weight or priority level (if applicable), its default value, a vector of Filter objects that it uses for packet classification, and m_queue, the packet queue itself.

In order for a packet to be classified to a specific TrafficClass, it must match at least one of the filters in its filters vector. Filter is another added class that holds a vector of FilterElements called elements. It matches packets only when a packet meets the criteria for all FilterElements in elements. FilterElement itself is an abstract base class designed for flexible packet matching. It holds a match criteria, which must be implemented per subclass. The FilterElement subclasses that have been implemented for this design include: SrcIPAddress, SrcMask, SrcPortNumber, DstIPAddress, DstMask, DstPortNumber, ProtocolNumber.

Thus, instantiating multiple TrafficClass objects gives a QoS multiple queues to pull packets from based on the collection of FilterElements and Filters added to the TrafficClass. A very high level overview of this base class design is included in Figure 1.

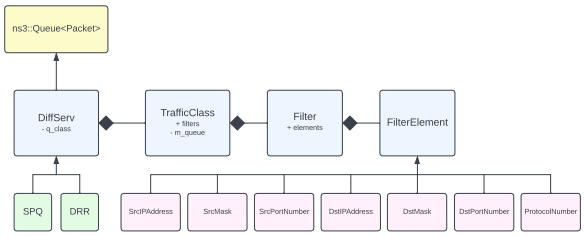

Figure 1: Base Class Design

Since DiffServ is a subclass of ns3's Queue class it makes use of and overrides the following Queue class methods: Enqueue, Dequeue, Remove, and Peek. In this design, each of these methods simply make a call to DoEnqueue, DoDequeue, DoRemove, and DoPeek, respectively. DiffServ's additional public methods that are not included in it's parent class include Schedule, Classify, and AddQueue. Each of these methods are discussed in detail in the following sections. We also have the GetQueues protected method for subclass to easily interact with q_class.

2.1 Classify

Classify sorts the traffic packets into their corresponding traffic queues by using the filters for each queue in q_class. It is called by DoEnqueue and returns the index of the matched queue in q_class. If a packet does not match any of the queues and a default queue is defined, it will be classified into the default queue. If a default queue has not been defined, the packet will be dropped. Classify is able to be overridden by its subclasses to offer flexibility for specific QoS classifications. For the implementations of SPQ and DRR, it was not necessary to override Classify.

Note that it is up to the user to define a default queue properly. If a default queue is not set, then packets that cannot be classified will be dropped, which may be intended. If multiple default queues are set, then the last queue added will act as the default queue, which may not be intended.

2.2 Schedule

Schedule is the one pure virtual method of DiffServ that makes it an abstract class. It must be implemented by the subclass. It follows a specific QoS algorithm to schedule which traffic queue is to be served next. It does no dequeuing itself, instead it simply peeks into the next queue to be

served and returns a const packet. This way, DoDequeue, DoRemove, and DoPeek are all able to call on Schedule. This reduces the need to have to implement a separate method from Schedule that DoPeek would need to call if Schedule were to also dequeue the packet. Instead, all that needs to be done is for DoDequeue and DoRemove to then call on Classify with the packet from Schedule. Classify will then return the index of the queue from which the packet needs to be dequeued or removed from.

2.3 AddQueue

AddQueue is another public method that has been added. It simply gives the user an interface to add TrafficClass objects to q_class. Note that queues are not instantiated at the creation of a DiffServ subclass, instead they can be added later. This gives users more flexibility in when the creation of their queues may happen. For example, we may decide to add additional queues later on after the program has been running for some time. It has also been marked as a virtual class method to give subclasses the option to override it if needed. This is particularly useful for classes that, after adding a queue, may need to perform additional actions on their own class variables. This is seen in our DRR implementation discussed in section 4.3.

3 DESIGN CHALLENGES

One of the biggest challenges in designing this base class was deciding what Schedule should do. As described earlier, there are many trade offs between having Schedule return a dequeued packet or a peeked at packet. If Schedule were to dequeue, then extra care is needed for DoPeek. If DoPeek can no longer call on Schedule because it dequeues a packet, then another public method may be necessary for the subclasses of DiffServ to implement. However, this additional method would still need to follow the same QoS logic that Schedule is following to dequeue in the first place. This seems redundant and contradictive to the goal of creating a base class.

3.1 DiffServ Improvements

Redesigning Schedule to instead return the index of the queue to be next served instead of a packet (similar to how Classify returns the index of the queue that a packet is classified to) would help with the challenge described above. While our current design of having Schedule only peek at the next packet reduces redundancy, it results in implementation complexity. A lot of care is needed to deal with the fact that since Peek calls on DoPeek which calls on Schedule, not only does the returned packet need to be const, but also all of these functions (and their helpers) need to be const since Peek is overridden form ns3's Peek in Queue.

This became an apparent issue when implementing DRR, since when a packet is dequeued, deficit values need to be updated. Since Schedule cannot update these values, this must happen elsewhere. The redesigning of Schedule to return an index would eliminate the need for any workaround. Instead, DoDequeue, DoRemove, and DoPeek would simply take this index from Schedule and dequeue, remove or peek as needed, with no const limitations. Of course, this solution would still require Dequeue to be overridden so that additional behavior (like updating deficits) may happen. However, these methods would no longer need to call on Classify to figure out which queue is next to be served, removed, or peeked from. This would lessen the amount of steps needed and it would lessen the const implementation complexity.

4 DIFFSERV SUBCLASSES

DiffServ's described design simplifies QoS mechanism implementations. First we describe our SPQ implementation which, due to DiffServ, requires little work, and its API usage. Next, we describe DRR's implementation and API usage.

4.1 SPQ Implementation

SPQ's implementation only needs for us define Schedule, which is required for all DiffServ subclasses. It first retrieves the q_class queues by calling DiffServ's protected method GetQueues. It then simply loops through these queues to find the highest prioritized non empty queue to serve next. Once this is determined, it peeks and returns the first packet in the selected queue. All other traffic queue functionalities that are required of SPQ are taken care of in DiffServ.

4.2 SPQ API Usage

To create an actual SPQ instance, our simulation program requires the user to pass in an xml file with the necessary configurations. We classify queue priority by the destination port of the packets. For example, packets going to port 9000 may be marked high priority while packets going to port 7000 are marked low priority. Thus, a filter element DstPortNumber will be instantiated from the port info provided by the user. This filter element will be added to a Filter object, which will then be added to a TrafficClass object.

Each TrafficClass object will be instantiated and modified using the user's xml file data. This data includes the max packets allowed for the queue, its priority, and whether or not it is a default queue. The TrafficClass objects will then be added to our SPQ instance. An example of a properly defined xml file for SPQ can be seen below:

```
<QoS>
    <Name>SPQ</Name>
    <Queue no="1">
        <MaxPackets>12000</MaxPackets>
        <Priority>2</Priority>
        <Default>true</Default>
        <DestPort>7000</DestPort>
    </Queue>
    <Queue no="2">
        <MaxPackets>12000</MaxPackets>
        <Priority>1</Priority>
        <Default>false</Default>
        <DestPort>9000</DestPort>
    </Queue>
</QoS>
```

Note that the name of the QoS must be provided so that the simulation can properly create the differentiated service and its filters. Also, the lower the priority number, the higher the priority, i.e. a queue with a priority of 1 is higher than one with a priority of 2.

4.3 DRR Implementation

The implementation for DRR requires a bit more modification than SPQ. For DRR, Schedule, Dequeue, and AddQueue need to be implemented. For Schedule, the algorithm implemented follows the steps described in Shreedhar and Varghese's Efficient Fair Queuing using Deficit Round Robin paper [1] with a few minor adjustments to account for the way ns3 dequeues packets (one peek for every interPacketInterval). Additionally, since Schedule is a const method and only peeks at the packet, the algorithm was adjusted so that no changes are applied to the deficits.

DRR has four private variables: activeQueue, nextActiveQueue, deficitCounter and nextDeficitCounter. The variables activeQueue and deficitCounter hold the index of the current queue and the current deficit values for each queue, respectively. Each time Schedule is called, nextActiveQueue and nextDeficitCounter will be set to activeQueue and deficitCounter and then updated as the algorithm runs. Resetting these variables to activeQueue and deficitCounter at the start of each Schedule call ensures that one may peek at a packet as many times as they like.

Since Schedule only takes a peek at the next packet and does no actual dequeuing, if any additional operations are required after a packet is dequeued (or removed), without adding additional public methods to DiffServ, there is no obvious way for these operations to happen. Changing Schedule to dequeue packets instead of just peeking will add additional methods and overrides necessary for a subclass to implement for DoPeek since it requires a const method for its implementation. This complicates subclass implementation, and thus, the best approach is to override Dequeue.

Only when Dequeue is called will activeQueue and deficitCounter be set to the next values. This is why Dequeue must be overridden. It will first call on DiffServ's Dequeue, and subsequently, Diffserv's DoDequeue and then make these necessary changes. The next implemented method, AddQueue, simply calls on DiffServ's AddQueue and then adds a new index to deficitCounter set to 0. These are the only implementation changes required for DRR.

4.4 DRR API Usage

Like SPQ, DRR requires an xml file with user defined values for its initial setup. It also filters the queues by destination port numbers, where each destination port number is associated with a specific weight. The construction of the FilterElement, Filter, and TrafficClass follows the same steps as described in SPQ, except this time weight is required instead of priority. An example of a properly defined xml file for DRR can be seen below:

```
<QoS>
    <Name>DRR</Name>
    <Queue no="1">
        <MaxPackets>12000</MaxPackets>
        <Weight>100</Weight>
        <Default>true</Default>
        <DestPort>6000</DestPort>
    </Queue>
    <Queue no="2">
        <MaxPackets>12000</MaxPackets>
        <Weight>200</Weight>
        <Default>false</Default>
        <DestPort>7000</DestPort>
    </Queue>
    <Queue no="3">
        <MaxPackets>12000</MaxPackets>
        <Weight>300</Weight>
        <Default>false</Default>
        <DestPort>9000</DestPort>
    </Queue>
</QoS>
```

Note that it is up to the user to define weights that are above 0, otherwise DRR will not work as intended.

5 SIMULATION RESULTS

Using the above xml file templates and values, we run two ns-3 simulations to validate our SPQ and DRR implementations. For both simulations we have a 3 node topology with data

rates set to 4-1 Mbps. The middle node is a QoS enabled router while the end nodes are UDP client and UDP server applications.

5.1 SPQ Validation

For SPQ, the client and server nodes each have two UDP applications. The server's applications are running on the ports described in the xml file (7000 and 9000). The lower priority application on the client begins sending UDP packets to port 7000. Around 13 seconds later, the higher priority application begins sending packets to port 9000. Figure 2 shows the packets that enter the router. When both applications are sending packets, we can see that the available bandwidth is shared equally between the two.

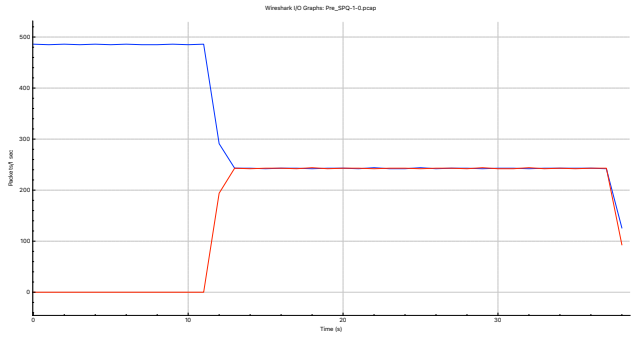

Figure 2: Pre SPQ Validation

When the packets leave the router, before the high priority packets are sent, the lower priority packets have all the bandwidth. However, once the high priority packets come in, the lower priority packets are no longer served by the router, instead the router serves only the high priority queue. We see this happen at around 13 seconds in Figure 3, which plots the packets as they are leaving the router. Once the high priority packets are done being sent, the low priority packets are able to resume being served by the SPQ router, seen at around 16 seconds.

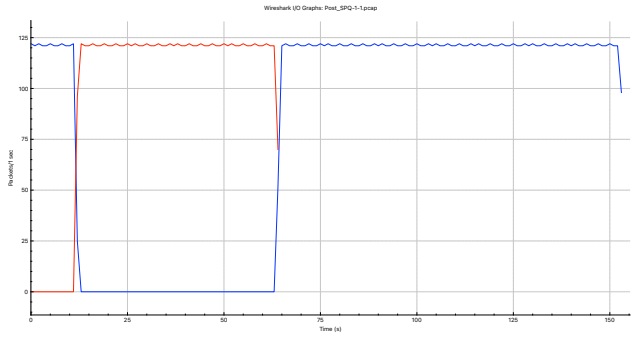

Figure 3: Post SPQ Validation

5.2 DRR Validation

For DRR, the client and server nodes each have three UDP applications. As in SPQ, the server's applications are running on the ports specified by the xml file (6000, 7000, and 9000 with weights 100, 200, and 300 respectively). This time, however, all client applications begin sending packets at the same time. We can see that the bandwidth is shared equally among these three applications as they enter the router, as shown in Figure 4.

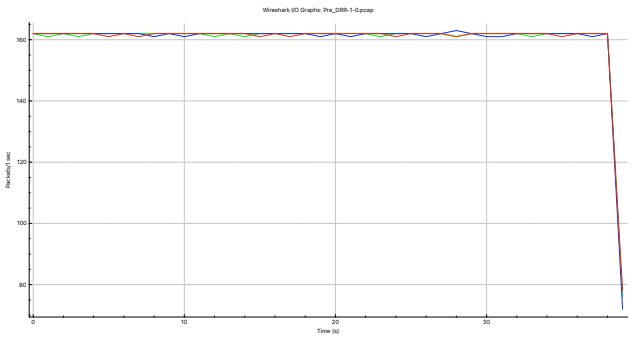

Figure 4: Pre DRR Validation

In Figure 5, as packets leave the router, they share the bandwidth in a 3:2:1 ratio. Each line represents a different destination port; red is port 9000 with weight 300, blue is port 7000 with weight 200, and green is port 6000 with weight 100. As the queues with weight 300 finish sending packets we see the two lower weighted queues gain more bandwidth, however, note that they still maintain their 2:1 ratio. Once the packets from the queue with weight 200 finish, the packets from the 100 weighted are able to utilize the entire available bandwidth.

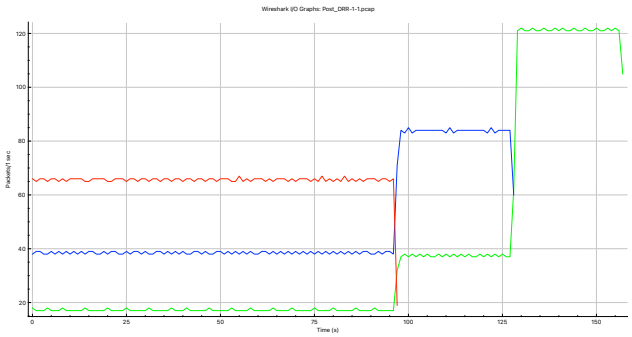

Figure 5: Post DRR Validation

6 CONCLUSION

The above simulations demonstrate that both SPQ and DRR implementations are effective and valid. Thus, our DiffServ

base class is able to provide the core functionalities required to simulate differentiated services. Furthermore, DiffServ provides us with a simple, effective, and non redundant interface where we can easily implement QoS mechanisms. Using the FilterElement and Filter building blocks from our design, we can easily create more complex boolean expressions for packet filtering and extend the functionality of DiffServ.

REFERENCES

[1] George Varghese M. Shreedhar. 1996. Efficient Fair Queuing using Deficit Round Robin. (1996), 234–236.

[2] ns3. 2014. ns3::Queue Class Reference. https://www.nsnam.org/docs/release/3.19/doxygen/classns3_1_1_queue.html. Accessed: 2023-05-10.