A Base Class to Simulate Differentiated Services

Lucas Palmer
University of San Francisco

ABSTRACT

Differentiated services is a class of service model for specifying and controlling IP network traffic by class. Packets that pass through a network device, such as a router, have their relative priority differentiated from one another. A base class will be used to hold the shared structure amongst differentiated services, while subclasses can implement different QoS algorithms.

1 DESIGN

The DiffServ class is a subclass of ns3::Queue<ns3::Packet>. The functions Enqueue(), Dequeue(), Remove(), and Peek() are overridden as private in the DiffServ class. These functions are left to the ns3 framework that utilizes the queue as a ns3::Queue<ns3::Packet>. Instead, DiffServ has three public pure virtual functions that each QoS base class must override: Schedule(), ScheduleConst(), and Classify().

The project UML diagram is used as is with the following modifications:

1. The DiffServ q_class member is protected so that subclass QoS implementations can enqueue and dequeue packets in each traffic class.

2. The DiffServ Classify() method returns a boolean that indicates whether a packet was enqueued into a traffic class.

3. The public method ScheduleConst() is added to DiffServ to obtain the next scheduled packet without any modifications to underlying traffic classes or QoS implementation state.

4. Include Peek(), Empty(), GetWeight(), GetPriority(), and IsDefault() public methods in the TrafficClass class, so that QoS implementations can enqueue and dequeue packets.

5. Combine SourceIpAddress and SourceMask filter elements into one filter, since a mask equality check requires a corresponding address. When only an IP address is provided, the all-ones mask can be used by default. Same with DestinationIpAddress and DestinationMask.

6. In each aggregation relationship between classes, use std::unique_ptr in the aggregate (parent) class to maintain ownership over the assembly (child) class.

The constructor of each aggregate class will take an XML node as an argument. Attributes of the aggregate class will be stored in XML node attributes, whereas the configuration of each assembly class will be in child XML nodes. It is the aggregate class' responsibility to iterate over child XML nodes and pass these XML configurations to the constructor of the right assembly class.

If a QoS implementation needs to maintain state in between calls to Schedule (e.g., DRR), the QoS subclass of DiffServ is responsible for maintaining such state with any required invariants.

2 DESIGN CHALLENGES

Constructors were designed to take XML nodes as arguments because the aggregate relationships between classes are analogous to an XML node hierarchy. It was desired for each class to own its own XML attribute/node parsing, so that a single XML configuration could be passed to the higher aggregate class (DiffServ) and propagate down to each assembly class. Although this worked as intended, input validation became a challenge. What if the XML configuration passed into a constructor does not have the required attributes or child nodes? In future work, an exception should be thrown if input is invalid. Otherwise, it is easy to misconfigure something like a traffic class (e.g., with a weight of 0 in deficit round robin).

Some QoS algorithms do not need to be stateful in between calls to Schedule(). For example, each call to SPQ is the same regardless of what packet was previous sent. Schedule() can just find the highest priority packet available to send. However, other QoS algorithms do need to be stateful in between calls to Schedule(). One such algorithm is DRR as described in http://cs621.cs.usfca.edu/v/resources/drr.pdf. The algorithm here is to iterate over each traffic class, allocating additional bytes to the traffic class each round and attempting to send a packet if allowed. This means that a subsequent call to Schedule() must know which traffic class is under consideration next. Should this state be stored in the QoS implementation subclass of DiffServ, or in the TrafficClass class? The former would mean that the QoS implementation is owning and maintaining metadata for each TrafficClass, whereas the latter would mean that TrafficClass further diverges from the more general definition in the project specification. Maintaining state in the QoS implementation subclass of DiffServ gives rise to several invariants for the lifetime of the class (e.g., the list of active traffic classes must have non-empty queues). In future work, such invariants should be better documented and violations should throw exceptions.

3 IMPLEMENTATION DETAILS

3.1 Queue Configuration

Queue configuration is stored in XML. An example configuration for SPQ is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<spq_config>
  <traffic_class priority_level="9">
    <filter>
      <filter_element>
        <source_ip ip="10.1.1.1"></source_ip>
      </filter_element>
      <filter_element>
        <dest_port port="9"></dest_port>
      </filter_element>
    </filter>
  </traffic_class>
  <traffic_class priority_level="10">
    <filter>
      <filter_element>
        <source_ip ip="10.1.1.1"></source_ip>
      </filter_element>
      <filter_element>
        <dest_port port="10"></dest_port>
      </filter_element>
    </filter>
  </traffic_class>
</spq_config>
```

This means that there are two traffic classes with priority 9 and 10. The traffic class with priority 9 is for traffic from source IP 10.1.1.1 to destination port 9. The traffic class with priority 10 is for traffic from source IP 10.1.1.1 to destination port 10. An example configuration for DRR is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<drr_config>
  <traffic_class weight="1">
    <filter>
      <filter_element>
        <source_ip ip="10.1.1.1"></source_ip>
      </filter_element>
      <filter_element>
        <dest_port port="9"></dest_port>
      </filter_element>
    </filter>
  </traffic_class>
  <traffic_class weight="2">
    <filter>
      <filter_element>
        <source_ip ip="10.1.1.1"></source_ip>
      </filter_element>
      <filter_element>
        <dest_port port="10"></dest_port>
      </filter_element>
    </filter>
  </traffic_class>
  <traffic_class weight="3">
    <filter>
      <filter_element>
        <source_ip ip="10.1.1.1"></source_ip>
      </filter_element>
      <filter_element>
        <dest_port port="11"></dest_port>
      </filter_element>
    </filter>
  </traffic_class>
</drr_config>
```

This means that there are three traffic classes with weight 1, 2, and 3. The traffic class with weight 1 is for traffic from source IP 10.1.1.1 to destination port 9. The traffic class with weight 2 is for traffic from source IP 10.1.1.1 to destination port 10. The traffic class with weight 3 is for traffic from source IP 10.1.1.1 to destination port 11.

3.2 Strict Priority Queueing

SPQ does not need to maintain state in between calls to Schedule(). On a call to Schedule(), the highest priority packet is sent regardless of what packet was previously sent. The implementation of SPQ in a QoS subclass of DiffServ is relatively simple. In the constructor of the subclass when traffic classes are added from an XML configuration, sort the traffic classes in decreasing order of priority. Schedule() will then iterate through the list of traffic classes and return the first packet it finds.

3.3 Deficit Round Robin

DRR does need to maintain state in between calls to Schedule(). Introduce the following members to the QoS subclass of DiffServ:

```
struct ActiveTrafficClass
{
    TrafficClass* traffic_class;
    double_t deficit_counter;
};

private:
  std::queue<ActiveTrafficClass> active_list_;
  std::unordered_set<TrafficClass*> active_set_;
```

The queue is to maintain the order of active traffic classes with their corresponding deficit counters. The unordered set is for fast lookup of whether or not a traffic class is in the active list.

In DeficitRoundRobin::Classify(), check for a matching traffic class and enqueue accordingly. In addition, add the traffic class to the back of the active list queue with a deficit counter of 0 only if it is not already in the queue.

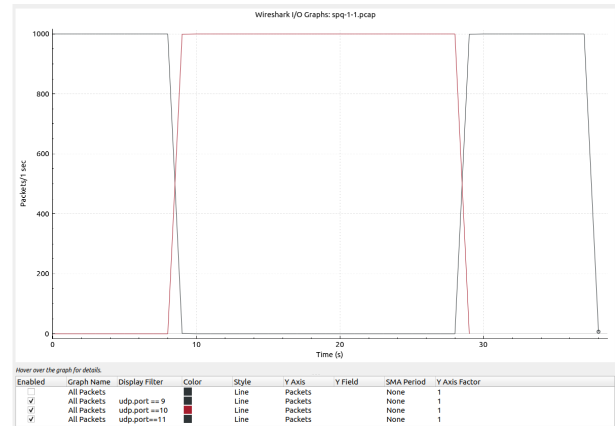
In DeficitRoundRobin::Schedule(), start with looking at the traffic class at the front of the active list queue (this

traffic class should be non-empty if the active list invariant is properly maintained). Do not pop this traffic class out of the queue yet, as it is possible that this class will stay at the head of the queue for the next call to Schedule(). Follow DRR by adding a quantum to the deficit counter. If there are not enough bytes in the deficit counter to send the next packet, push the traffic class to the back of the active list queue. If there are enough bytes in the deficit counter to send the next packet, this is the packet that will be returned on this call to Schedule() and the deficit counter is decremented by the number of bytes in the packet. However, we can not just push this traffic class to the back of the queue for the next call to Schedule(). If the deficit counter is still enough to also send the next packet in the same traffic class, keep the head of the active list queue the same. *One subtlety is that, if we leave a traffic class at the head of the active list queue, we don't want to add another quantum to the deficit counter on the next call to Schedule(). For simplicity in this implementation, just subtract a quantum from the deficit counter prior to returning a packet so that adding a quantum on the next call is negated.*

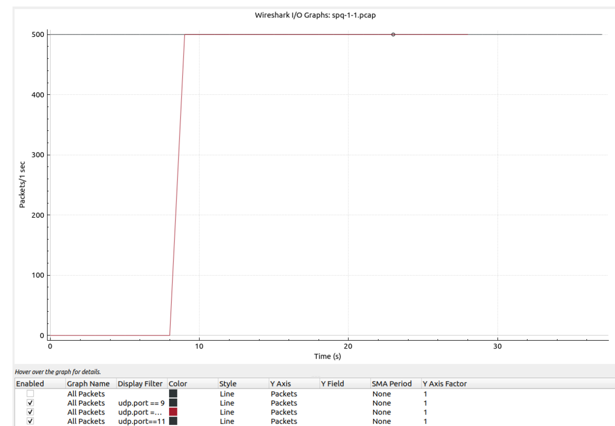
4 SIMULATIONS

4.1 Strict Priority Queueing

To validate SPQ, configure a 3 node topology (client, router, server) with point-to-point connections client->router and router->server. Set the data rate of client->router to 40Mbps and the data rate of router->server to 10Mbps (to create a bottleneck). Add the custom queue implementation to the router net device on the router->server point-to-point connection (use the XML configuration included in section 3.1). Configure two bulk UDP applications to send a stream of UDP packets from the client node to ports 9 and 10 on the server node. For this demonstration, we want to show the higher priority traffic to port 10 exhausting the bottleneck. One bulk UDP application is configured to send 10Mbps of traffic to port 9 from time 1.0 to time 39.0. The second bulk UDP application is configured to send 10Mbps of traffic to port 10 from time 10.0 to time 30.0. *Note that 10Mbps = 1250000 bytes/second so the applications send 1250 byte packets at a rate of 1000 times per second.*

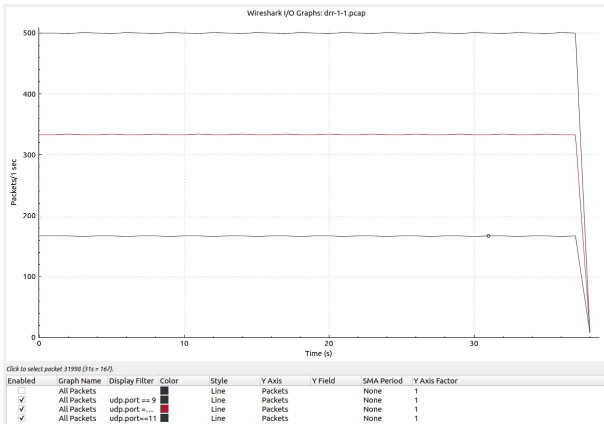


Halve the rate of sending packets, and the bottleneck is big enough for all traffic to get through.

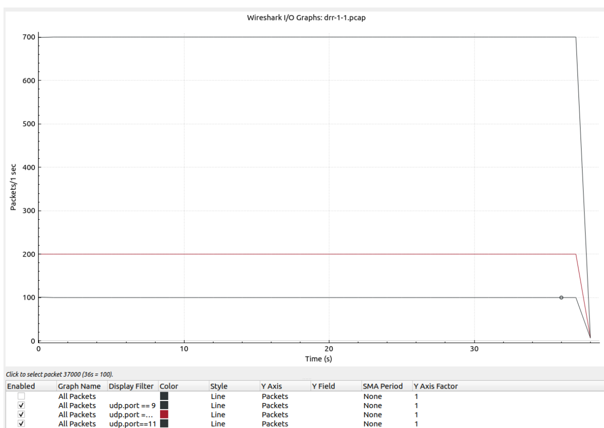


4.2 Deficit Round Robin

To validate DRR, configure a 3 node topology (client, router, server) with point-to-point connections client->router and router->server. Set the data rate of client->router to 40Mbps and the data rate of router->server to 10Mbps (to create a bottleneck). Add the custom queue implementation to the router net device on the router->server point-to-point connection (use the XML configuration included in section 3.1). Configure three bulk UDP applications to send a stream of UDP packets from the client node to ports 9, 10, and 11 on the server node. For this demonstration, we want to show the the ratio of traffic when no traffic class can get transmitted entirely. One bulk UDP application is configured to send 10Mbps of traffic to port 9 from time 1.0 to time 39.0. The second bulk UDP application is configured to send 10Mbps of traffic to port 10 from time 1.0 to time 39.0. The third bulk UDP application is configured to send 10Mbps of traffic to port 11 from time 1.0 to time 39.0. *Note that 10Mbps = 1250000 bytes/second so the applications send 1250 byte packets at a rate of 1000 times per second.*



Changing the quantum values from 1:2:3 to 1:2:7 shows the same ratio in packets.



5 API USAGE

The constructors of the queue subclasses take a const reference to rapidxml::xml_node<>. In order to use these classes, parse an XML config in the expected format (see 3.1) using rapidxml and pass the result to the constructor.

```
std::string config_file = "myfile.xml";
rapidxml::file<> xmlFile(config_file.c_str());
rapidxml::xml_document<> doc;
doc.parse<0>(xmlFile.data());

xml_node<>* spq = doc.first_node("spq_config");
if (spq != nullptr)
{
    StrictPriorityQueueing queue(*spq);
}

xml_node<>* drr = doc.first_node("drr_config");
if (drr != nullptr)
{
    DeficitRoundRobin queue(*drr);
}
```

6 ALTERNATIVE DESIGN

The current design is quite rigid in that all configuration is done in constructors. It would be useful to provide a default constructor with a public interface for updating traffic classes dynamically.

7 REFERENCES

Rapidxml (https://rapidxml.sourceforge.net/) was used as an XML parser in this project. DRR was implemented as described in http://cs621.cs.usfca.edu/v/resources/drr.pdf.