A Base Class to Simulate Differentiate Services

Vahab Pournaghshband[*]
Aren Mark Boghozian[+]
Dam Ju Kim[+]

The Advanced Network and Security Research Laboratory
University of San Francisco[*]
California State University, Northridge[+]

IARIA

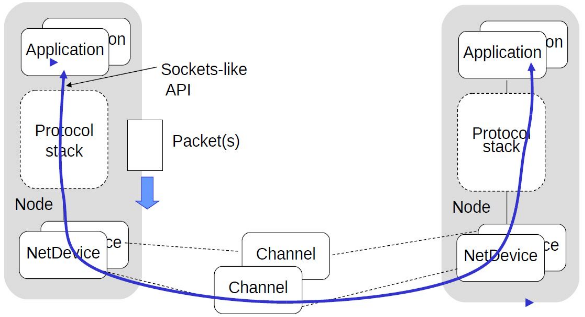
The 15th International Conference
on Networking and Services
ICNS 2019
June 2- 6, 2019- Athens, Greece

Introduction

- Implementing and simulating differentiated services from scratch creates redundancy in workflow, as differentiated services follow similar structure and share many features.
- We are first to propose a base class that is designed to eliminate the redundancy issue by simplifying the implementation to the core algorithms, while the base class provides all necessary apparatus.
- The simplicity and ease of use of the base class eliminates the redundancy involved in implementing the common features of differentiated services.

Network Simulator 3 (ns-3)

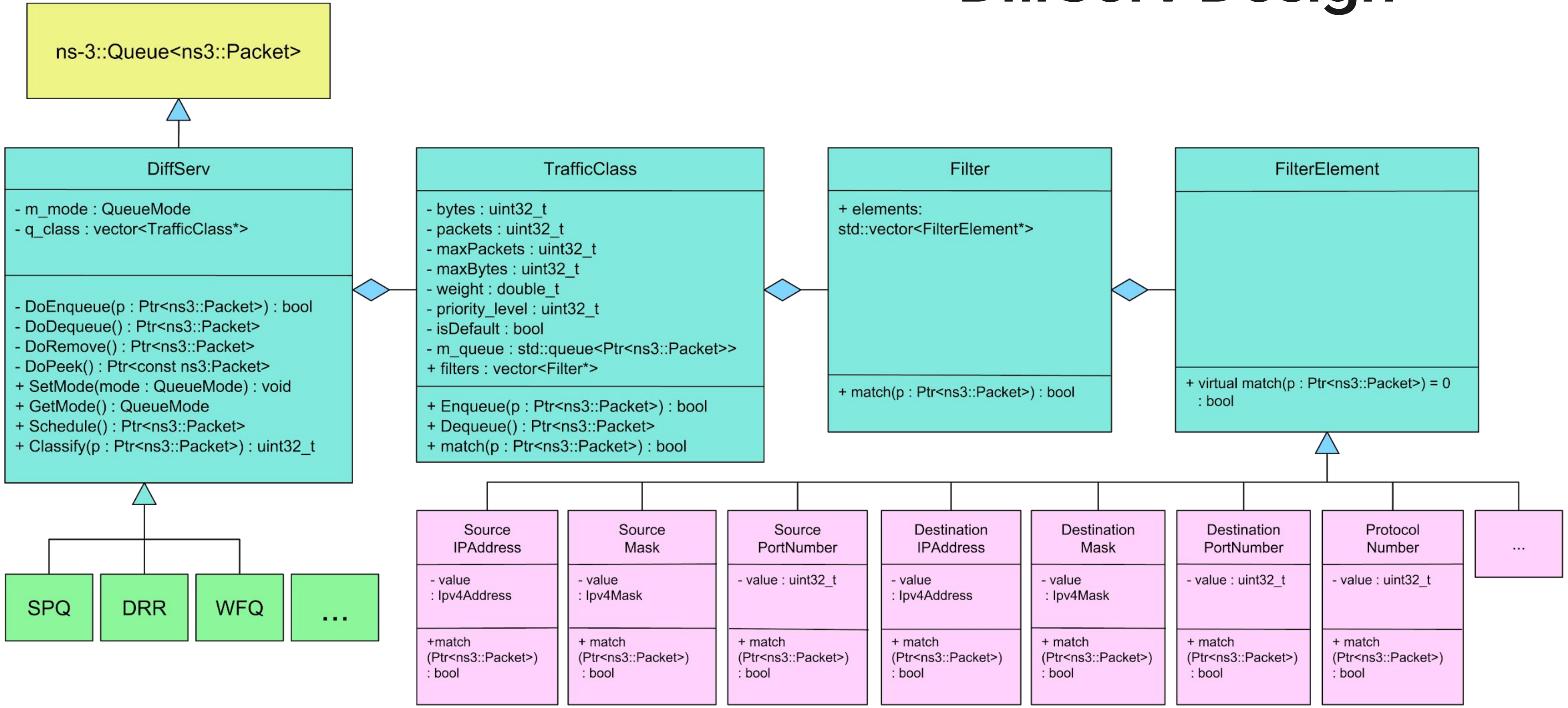
- A discrete-event open source simulator for network research and education
- ns-3 core is written entirely in C++
- ns-3 simulates networking architecture in Linux
- Complex simulation features include
  - Extensive parametrization system
  - Configurable embedded tracing system
  - Standard outputs to text logs or PCAP (tcpdump/wireshark)
- Object oriented design for rapid coding and extension
  - Automatic memory management
  - Object aggregation/query for new behaviors and state
    - E.g., adding mobility models to nodes
- Models true IP stack, with potentially multiple devices & IP addresses on every node



DiffServ Class

- Base class provides basic functionalities required to simulate differentiated services
    - Classification
        - Classify function utilizes filter aspect to sort the traffic packets into appropriate traffic queues.
    - Scheduling
        - Schedule function carries out designed Quality-of-Service (QoS) algorithm to schedule which traffic queue to be served at the time.
- Two separate functions need to be implemented per QoS algorithm design of the developer’s choice.
- For network queues, DoEnqueue() and DoDequeue() functions can be overwritten to meet implementation requirements for various QoS algorithms.

DiffServ Design


ns-3::Queue<ns3::Packet>
DiffServ
- m_mode : QueueMode
- q_class : vector<TrafficClass*>
- DoEnqueue(p : Ptr<ns3::Packet>) : bool
- DoDequeue() : Ptr<ns3::Packet>
- DoRemove() : Ptr<ns3::Packet>
- DoPeek() : Ptr<const ns3:Packet>
+ SetMode(mode : QueueMode) : void
+ GetMode() : QueueMode
+ Schedule() : Ptr<ns3::Packet>
+ Classify(p : Ptr<ns3::Packet>) : uint32_t
TrafficClass
- bytes : uint32_t
- packets : uint32_t
- maxPackets : uint32_t
- maxBytes : uint32_t
- weight : double_t
- priority_level : uint32_t
- isDefault : bool
- m_queue : std::queue<Ptr<ns3::Packet>>
+ filters : vector<Filter*>
+ Enqueue(p : Ptr<ns3::Packet>) : bool
+ Dequeue() : Ptr<ns3::Packet>
+ match(p : Ptr<ns3::Packet>) : bool
Filter
+ elements: std::vector<FilterElement*>
+ match(p : Ptr<ns3::Packet>) : bool
FilterElement
+ virtual match(p : Ptr<ns3::Packet>) = 0 : bool
SPQ
DRR
WFQ
...
Source IPAddress
- value : Ipv4Address
+match (Ptr<ns3::Packet>) : bool
Source Mask
- value : Ipv4Mask
+ match (Ptr<ns3::Packet>) : bool
Source PortNumber
- value : uint32_t
+ match (Ptr<ns3::Packet>) : bool
Destination IPAddress
- value : Ipv4Address
+ match (Ptr<ns3::Packet>) : bool
Destination Mask
- value : Ipv4Mask
+ match (Ptr<ns3::Packet>) : bool
Destination PortNumber
- value : uint32_t
+ match (Ptr<ns3::Packet>) : bool
Protocol Number
- value : uint32_t
+ match (Ptr<ns3::Packet>) : bool
...

Traffic Class

- Defines characteristics of a queue
  - Mode
  - Queue priority level
  - Max bytes
  - Max Packets
  - Weight
- Provides functionality to check user predefined criteria
  - Match(Packet) : bool
  - Match function at *TrafficClass* level provides OR comparison of all filters in its vector’s match return value.
- Can be configured to have N number of unique or similar queues

Filters and Filter Elements

- Filter
  - Filter class provides a vector of FilterElements
  - match function at *filter* level provides AND comparison of each FilterElement in its vector's match return value
- Layered approach of match function provides detailed network traffic characterization.

- Filter Elements
  - Allows User to define network traffic properties
  - IP address
    - Source IP
    - Destination IP
  - Port number
    - Source Port Number
    - Destination Port Number
  - Protocols
    - TCP
    - UDP
    - ...

API

- Easy to use API, makes for a simple QoS implementation
  - Specify and retrieve the operative mode of the device (packet or byte base)
    - SetMode(mode)
    - GetMode()
  - Classify incoming packets into queue
    - Classify(Packet)
    - DoEnqueue(Packet)
    - Enqueue(Packet)
  - Schedule and forward packets into outgoing interface
    - Schedule()
    - DoDequeue()
    - Dequeue()
  - Initialize as many number of queues with unique characteristics
    - Q_class: vector<TrafficClass*>

API Usage

- Instantiate the Queues in constructor
  - `StrictPriorityQueue::StrictPriorityQueue() : BaseClass()`
  - `q_class.push_back(new TrafficClass());`
  - `q_class[0]->setMaxPackets(100);`
- Classify packets into appropriate queues

  -
    ```
    uint16_t StrictPriorityQueue::classify (Ptr<QueueItem> p)
      if(q_class[0]->filters[0]->element[0]->match(p))
          return 1;  //Priority Level
    ```
- Insert packet into the queue

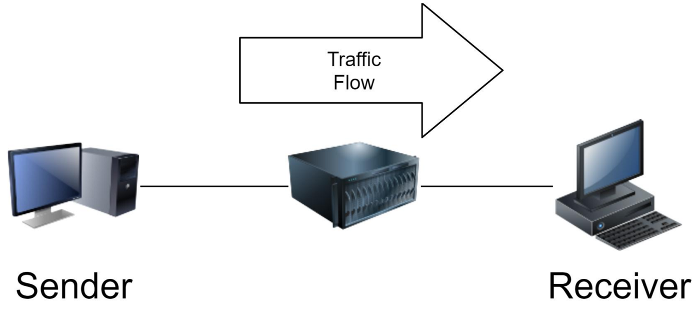
  -
    ```
    bool StrictPriorityQueue::DoEnqueue (Ptr<QueueItem> p)
       if(priority == 1) q_class[0]->m_queue.push(p);
    ```
- Determine which traffic queue to be served

  -
    ```
    void StrictPriorityQueue::schedule(Ptr<QueueItem> p)
      if(!q_class[0]->m_queue.empty())
        DoDequeue();
    ```
- Dequeue the packet from queue and push into outbound interface

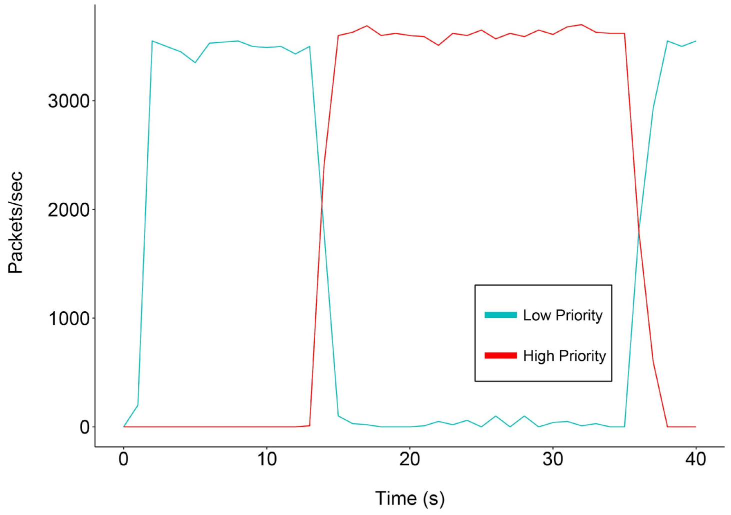
  -
    ```
        Ptr<ns3::Packet> p = q_class[0]->m_queue.front();
        q_class[0]->m_queue.pop();
    ```

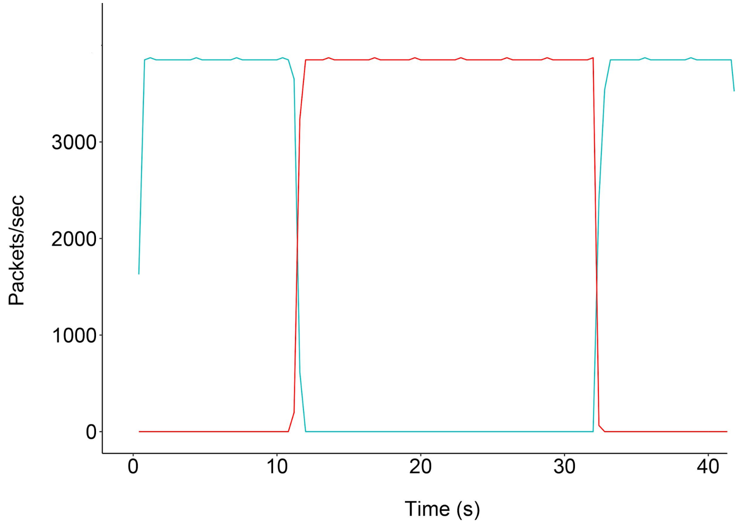
Validation Environment

- Both simulated and real validation environment consist of two machines; sender and receiver, connected by a switch with QoS capabilities.
- First, we ran series of experiments based on selected scenarios with predictable outcomes for a given set of parameters in our simulated environment. Then, we ran simulations of the scenario using the implemented modules and compared the recorded results with our analytic model.
- Second, we replicated a set of experiment scenarios in the real validation environment, and compared behavior and performance analysis between our simulated and real QoS-enabled switch.



Validation Results





The plots compare mean bandwidth allocation for real environment (left) and simulated environment in ns-3 (right). The QoS feature simulated in these environments is strict priority queueing. Both environments use identical network parameters. In this scenario, the sender starts sending low priority traffic at time 0s. At time 12s, the sender sends both high priority traffic. Once high priority traffic begins, low priority traffic bandwidth allocation is severely reduced. Once high priority traffic ends, then low priority traffic regains full available bandwidth allocation.

Conclusions

- The source code is publicly available at GitHub : https://github.com/vhbsoft/telesto
- The source code includes the base class, along with the implementation of Strict Priority Queueing and Deficit Round Robin using the base class.
- The *DiffServ* Design provides users with a simplified API for implementing various QoS methods.
- The simplicity and ease of use will eliminate the redundancy involved in implementing the common features of differentiated services.
- Will implement the base class design to other network simulators, such as Click.
- Base class will be extended to support different QoS algorithms and for further validations (e.g., Weighted Round Robin and Weighted Fair Queuing).
- *Acknowledgements:* the authors are grateful to Robert Chang for his contributions to the DiffServ class design.