Project 2 Report: Implementing SPQ and DRR in ns-3

Shijie Xu and Jinglin Zhou

ABSTRACT

This report details the implementation of Strict Priority Queuing (SPQ) and Deficit Round Robin (DRR) scheduling algorithms within the ns-3 network simulator for Project 2 in CS621: Network Programming. We describe the design, implementation, and validation of these algorithms, discuss challenges and limitations, and propose improvements to the DiffServ framework. The report also covers alternative design considerations and references used.

KEYWORDS

Network Simulation, ns-3, DiffServ, SPQ, DRR

1 INTRODUCTION

In this project, we implemented two queue scheduling algorithms—Strict Priority Queuing (SPQ) and Deficit Round Robin (DRR)—within the ns-3 network simulator to enhance the Differentiated Services (DiffServ) framework. DiffServ provides an architecture for managing network traffic to ensure quality of service (QoS) by differentiating packets based on their service requirements [1]. SPQ prioritizes packets based on queue priority, ensuring low-latency for high-priority traffic, while DRR ensures fair bandwidth allocation using a weighted round-robin approach, mitigating starvation issues in high-priority scenarios. These algorithms are critical for managing network traffic in scenarios requiring QoS, such as multimedia streaming or real-time communications. The ns-3 simulator, a widely-used tool for network research [2], provides a robust platform for testing such network protocols in a controlled environment. This report outlines our design, implementation details, validation, challenges, and suggestions for improvement.

2 DESIGN

We created and modified several ns-3 classes to implement SPQ and DRR, leveraging the existing DiffServ framework. The design focuses on integrating these scheduling algorithms into the ns-3 network simulator, ensuring modularity, extensibility, and efficient packet handling. Below, we detail the architecture, class interactions, packet classification mechanisms, and configuration handling.

2.1 Architecture Overview

The SPQ and DRR implementations are built as extensions of the `DiffServ` class in ns-3, which provides a foundational framework for differentiated services. The architecture consists of three main components:

- **Scheduling Logic**: The core scheduling algorithms (SPQ and DRR) determine the order in which packets are dequeued from the router's queues.
- **Packet Classification**: A filtering mechanism assigns incoming packets to specific queues based on predefined criteria, such as destination ports.
- **Queue Management**: The `TrafficClass` class manages individual queues, maintaining state information like priority levels (for SPQ) or deficit counters (for DRR).

The `DiffServ` base class provides the infrastructure for queue management and packet classification, while SPQ and DRR override the scheduling behavior to implement their respective algorithms. The simulation scripts (`spq-simulation.cc` and `drr-simulation.cc`) set up the network topology and traffic patterns to test these implementations.

2.2 New and Modified Classes

- **DiffServ (diffserv.h, diffserv.cc)**: A base class providing core functionality for differentiated services, including queue management, packet classification, and configuration parsing. Modified to support SPQ and DRR by defining virtual methods `Classify` and `Schedule`, which are overridden by subclasses. Manages a vector of `TrafficClass` queues and implements `ReadConfigFile` to parse configuration files specifying queues and filters.
- **SPQ (spq.h, spq.cc)**: A new class derived from `DiffServ` to implement strict priority queuing. The `Schedule` method iterates through the queues in descending order of priority, selecting the first non-empty queue for dequeuing. The `Classify` method assigns packets to queues based on filter matches, ensuring that high-priority traffic (e.g., to port 9000) is processed before low-priority traffic (e.g., to port 7000).
- **DRR (drr.h, drr.cc)**: A new class derived from `DiffServ` for deficit round-robin scheduling. It maintains a `deficits` vector for each queue, incrementing the deficit by the queue's weight (quantum) in each round. The `Schedule` method dequeues packets from a queue only if its deficit is sufficient to cover the packet size, ensuring weighted fairness (e.g., a 3:2:1 bandwidth ratio for weights 300:200:100).
- **TrafficClass (traffic-class.h, traffic-class.cc)**: Modified to include additional attributes: `priority_level` for SPQ, which determines the queue's priority, and `weight` for DRR, which controls the quantum added per round. Getters and setters were added for these attributes, allowing the scheduling algorithms to access and modify queue properties dynamically.
- **Filter and FilterElement (filter.h, filter.cc, filter-element.h, filter-element.cc)**: New classes designed to support packet classification. The `Filter` class contains a list of `FilterElement` objects, each defining a criterion (e.g., destination IP, port, or protocol). The `Match` method in `Filter` evaluates whether a packet satisfies all criteria, enabling precise assignment to queues.
- **Simulation Scripts (spq-simulation.cc, drr-simulation.cc)**: Developed to test SPQ and DRR in a controlled network environment. These scripts configure a three-node topology (two hosts and one router) with

point-to-point links, generate UDP traffic at specified ports, and log packet traces for validation.

2.3 Packet Classification Mechanism

Packet classification is a critical component of the design, enabling the differentiation of traffic flows. The `Filter` class operates as follows:

- **Criteria Definition**: Each `FilterElement` specifies a single matching criterion, such as a destination port (e.g., 7000 for low-priority traffic in SPQ). Supported criteria include source/destination IP addresses, ports, and protocol types.
- **Matching Logic**: The `Filter::Match` method checks a packet against all `FilterElement` objects in the filter. A packet is assigned to a queue only if it matches all criteria defined in the filter.
- **Queue Assignment**: The `DiffServ::Classify` method iterates through a list of filters, each associated with a `TrafficClass` queue. The first filter that matches the packet determines the queue to which the packet is enqueued.

This mechanism ensures that packets are directed to the appropriate queue based on their characteristics, allowing SPQ to prioritize high-priority traffic and DRR to allocate bandwidth according to weights.

2.4 Class Interactions

The interaction between classes is designed to be modular and efficient:

- **Packet Enqueueing**: When a packet arrives at the router, the `DiffServ::DoEnqueue` method calls `Classify` to determine the target queue. The packet is then enqueued in the corresponding `TrafficClass` object.
- **Scheduling**: The `DiffServ::DoDequeue` method invokes the `Schedule` method, which is overridden by SPQ and DRR. For SPQ, `Schedule` selects the highest-priority non-empty queue. For DRR, it iterates through queues in a round-robin fashion, checking deficit counters to decide which queue to dequeue from.
- **Queue Management**: The `TrafficClass` class manages queue state, including the packet buffer (with a maximum size specified in the configuration), priority level (for SPQ), and deficit counter (for DRR). It provides methods like `IsEmpty` and `Dequeue` to support scheduling operations.

This modular design allows SPQ and DRR to share common functionality (e.g., packet classification) while implementing distinct scheduling behaviors.

2.5 Configuration Handling

The configuration files (`spq-config.txt` and `drr-config.txt`) provide a flexible way to define queues and filters dynamically:

- **File Format**: Each file contains lines specifying queues and filters. A queue line has the format `queue <id> <priority/weight> <maxPackets>`, where `<id>` is a unique identifier, `<priority/weight>` is the priority level (for SPQ) or weight (for DRR), and `<maxPackets>` is the maximum queue size. A filter line has the format `filter <queueId> <type> <value>`, linking a filter to a queue and specifying a criterion (e.g., `filter 0 destPort 7000`).
- **Parsing Logic**: The `DiffServ::ReadConfigFile` method reads the configuration file, parsing each line to create `TrafficClass` objects and associated `Filter` objects. For SPQ, it sets the `priority_level` of each queue; for DRR, it sets the `weight`.
- **Usage in Simulations**: The simulation scripts load the appropriate configuration file (e.g., `spq-config.txt` for SPQ simulations) and pass it to the `SPQ` or `DRR` instance, ensuring that the router's behavior is configured dynamically based on the specified parameters.

This configuration mechanism allows users to experiment with different queue setups and filter criteria without modifying the source code, enhancing the flexibility of the implementation.

2.6 Why and How

SPQ and DRR were implemented as subclasses of `DiffServ` to leverage its existing queue management and packet classification mechanisms, reducing code duplication and ensuring compatibility with ns-3's architecture. SPQ uses a simple priority-based scheduling algorithm, iterating through queues in order of priority to ensure high-priority traffic is served first. DRR introduces a deficit counter to ensure fairness, allowing each queue to send packets proportional to its weight in each round. Filters were designed to support flexible packet classification, enabling precise control over traffic differentiation. The configuration files provide a user-friendly interface for defining queues and filters, making the system adaptable to various network scenarios.

3 DESIGN ISSUES/CHALLENGES/LIMITATIONS

- **SPQ Starvation**: Low-priority queues may experience starvation if high-priority queues are constantly active, as seen in simulations where port 9000 traffic dominated.
- **DRR Complexity**: Managing deficit counters and ensuring fairness increased implementation complexity, particularly in handling edge cases like empty queues. The current design of the `Schedule` method, which returns both a queue index and a peeked packet, introduces additional complexity due to const-correctness constraints. Since `DiffServ::Peek` is a const method (as required by `ns3::Queue`), it calls `Schedule` via `DoPeek`, forcing `Schedule` and related methods to be const. This prevents `DRR::Schedule` from updating the `deficits` vector directly, requiring a workaround where deficit updates occur before scheduling, and the actual dequeue happens in `DiffServ::DoDequeue`.
- **Limited Filter Types**: The current filter implementation supports only basic criteria (IP, ports, protocol). More complex filters (e.g., DSCP fields) would enhance flexibility.
- **Simulation Scale**: The simulation scripts test a simple topology with three nodes. Larger networks may reveal additional performance issues.

Given more time, we would address starvation in SPQ with a minimum bandwidth guarantee for low-priority queues and optimize DRR's deficit management for efficiency. An alternative design consideration to reduce DRR's implementation complexity involves

redesigning the `Schedule` method to return only the index of the next queue to be served, similar to how `Classify` returns the queue index for enqueuing. Currently, `Schedule` returns a pair containing the queue index and a peeked packet, which requires `Schedule` to call `TrafficClass::Peek`. This creates a circular dependency since `DiffServ::Peek` calls `Schedule`, necessitating const-correctness across all related methods. Redesigning `Schedule` to return only the queue index would eliminate this dependency, allowing `Schedule` to be non-const and enabling `DRR::Schedule` to focus solely on selecting the next queue. The `DiffServ::DoDequeue`, `DoRemove`, and `DoPeek` methods would then use this index to dequeue, remove, or peek packets directly, removing the const constraint. For DRR, this redesign would allow deficit updates to occur in an overridden `DRR::Dequeue` method after dequeuing, simplifying the scheduling logic and eliminating the need for workarounds. While this approach might introduce minor redundancy (e.g., accessing the queue twice—once to check if it's empty in `Schedule` and again to dequeue), it significantly reduces implementation complexity and aligns with the modular design goals of the DiffServ framework.

4 SUGGESTIONS TO IMPROVE DIFFSERV

- **Modular Scheduling**: Introduce a scheduling interface in `DiffServ` to allow dynamic selection of algorithms (e.g., SPQ, DRR, WFQ) without subclassing.
- **Enhanced Filters**: Support advanced classification criteria like DSCP or QoS markings.
- **Statistics Collection**: Add built-in metrics (e.g., queue latency, packet drop rates) to `DiffServ` for easier performance analysis.
- **Configuration Validation**: Implement checks in `ReadConfigFile` to detect invalid or conflicting queue/filter configurations.

5 IMPLEMENTATION DETAILS

The SPQ implementation iterates over queues to select the highest-priority non-empty queue, logging scheduling decisions with timestamps. DRR maintains a `deficits` vector, incrementing each queue's deficit by its weight per round and scheduling packets only if the deficit is sufficient. Both classes read configurations from text files, parsing lines to create queues and filters. The simulation scripts set up a three-node topology (two hosts, one router) with point-to-point links (4Mbps host-to-router, 1Mbps router-to-host). UDP applications generate traffic at different ports, with SPQ prioritizing high-priority traffic (port 9000) and DRR allocating bandwidth based on weights (100, 200, 300).

6 SIMULATION VALIDATION AND VERIFICATION

We validated the SPQ and DRR implementations using a combination of ns-3's `FlowMonitor`, PCAP tracing, and Wireshark I/O graph analysis to ensure correct behavior of the scheduling algorithms.

6.1 Simulation Topology

The simulation setup consists of a three-node topology, as illustrated in Figure 1. It includes two end-hosts and one router connected via point-to-point links. The first link, between the sender (Source) and the router, has a data rate of 4 Mbps and a delay of 2 ms. The second link, between the router and the receiver (Destination), has a data rate of 1 Mbps (acting as the bottleneck) and a delay of 2 ms. This topology creates a controlled environment where the scheduling algorithms (SPQ and DRR) are applied at the router to manage traffic flows between the sender and receiver.

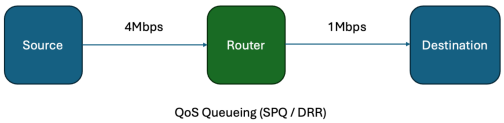

Figure 1: Three-node topology used in the SPQ and DRR simulations, consisting of two end-hosts (Host 0 and Host 2) and a router (Node 1) connected via point-to-point links.

6.2 SPQ Validation

For SPQ, the client and server nodes each ran two UDP applications, with server applications listening on ports 7000 (low priority) and 9000 (high priority), as specified in the configuration file. The low-priority application began sending UDP packets to port 7000 at the start of the simulation, achieving a rate of approximately 400 packets/sec. Around 13 seconds later, the high-priority application started sending packets to port 9000. Figure 2 shows the packet rate entering the router, where bandwidth is shared equally between the two applications when both are active, with each stream at around 200 packets/sec.

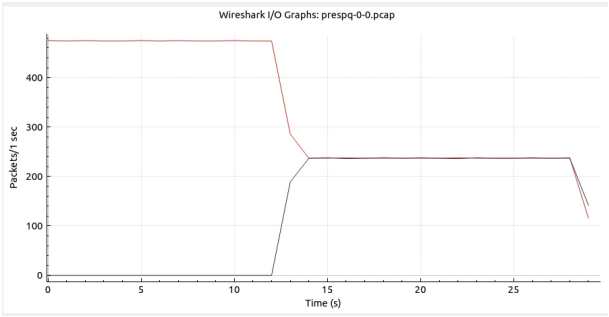

Figure 2: Pre-SPQ Validation: Packet rate entering the router (Wireshark I/O Graph from prespq-0-0.pcap).

Post-router, as shown in Figure 3, the router prioritizes high-priority packets (port 9000, gray line) starting at around 13 seconds, achieving a rate of approximately 125 packets/sec and completely starving the low-priority queue (port 7000, red line) until the high-priority traffic ceases at approximately 16 seconds. After this, the low-priority packets resume being served at around 125 packets/sec. PCAP traces confirmed correct packet ordering, and `FlowMonitor` data verified that high-priority traffic was scheduled first when both queues were active.

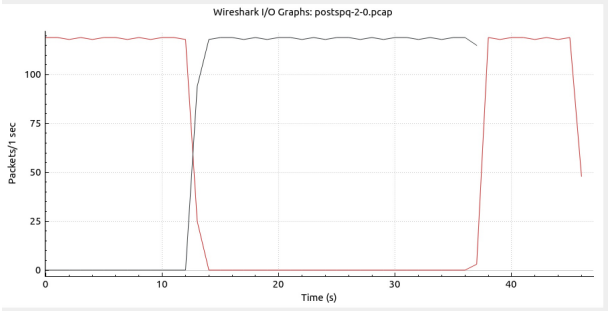

Figure 3: Post-SPQ Validation: Packet rate leaving the router (Wireshark I/O Graph from postspq-2-0.pcap).

6.3 DRR Validation

For DRR, the client and server nodes each ran three UDP applications, with server applications on ports 6000 (weight 100), 7000 (weight 200), and 9000 (weight 300), as defined in the configuration file. All client applications began sending packets simultaneously, each at a rate of approximately 150 packets/sec. Figure 4 illustrates the packet rate entering the router, showing equal bandwidth sharing among the three applications.

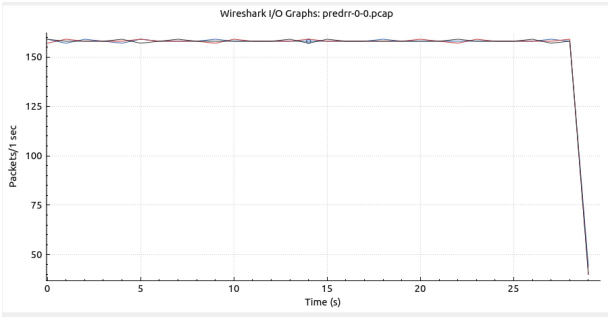

Figure 4: Pre-DRR Validation: Packet rate entering the router (Wireshark I/O Graph from predrr-0-0.pcap).

Post-router, as shown in Figure 5, the bandwidth is allocated in a 3:2:1 ratio corresponding to the weights (port 9000: gray, weight 300, around 60 packets/sec; port 7000: red, weight 200, around 40 packets/sec; port 6000: blue, weight 100, around 20 packets/sec). Around 40 seconds, the queue with weight 300 (port 9000) finishes sending packets, and the remaining queues (ports 7000 and 6000) adjust to a 2:1 ratio, with port 7000 at around 80 packets/sec and port 6000 at around 40 packets/sec. At approximately 45 seconds, the queue with weight 200 (port 7000) completes, allowing the queue with weight 100 (port 6000) to utilize the entire bandwidth at around 120 packets/sec. `FlowMonitor` data confirmed that port 9000 initially received approximately three times the bandwidth of port 6000, and logs verified correct deficit counter updates and round-robin scheduling.

6.4 Verification

Verification involved analyzing logs to ensure correct queue assignments, filter matches, and packet scheduling. Edge cases, such as empty or full queues, were tested to confirm robustness. The combination of `FlowMonitor` metrics (e.g., throughput, packet loss) and PCAP trace analysis validated the expected behavior of SPQ and DRR, with Wireshark I/O graphs providing visual confirmation of scheduling dynamics.

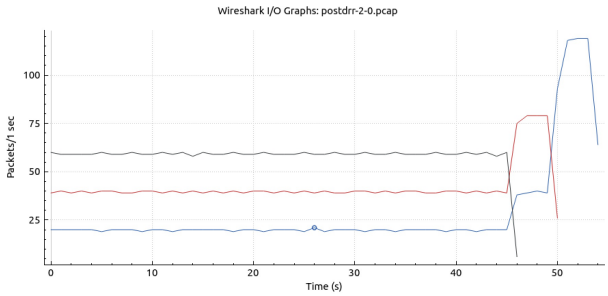

Figure 5: Post-DRR Validation: Packet rate leaving the router (Wireshark I/O Graph from postdrr-2-0.pcap).

7 API USAGE

To use SPQ or DRR, users must:

- Create an instance of `SPQ` or `DRR` and call `ReadConfigFile` with a configuration file specifying queues and filters.
- Assign the queue to a `PointToPointNetDevice` using `SetQueue`.
- Ensure traffic matches filter criteria (e.g., destination ports).

Configuration file format:

```
queue <id> <priority/weight> <maxPackets>
filter <queueId> <type> <value>
```

8 ALTERNATIVE DESIGN

If redesigning, we would:

- Use a single `DiffServ` class with pluggable scheduling modules to reduce code duplication.
- Implement a priority aging mechanism in `SPQ` to prevent low-priority starvation.
- Add a GUI-based configuration tool for easier queue and filter setup.
- Extend `TrafficClass` to support dynamic weight adjustments based on traffic conditions.

REFERENCES

[1] Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. 1998. *An Architecture for Differentiated Services.* RFC 2475. Internet Engineering Task Force (IETF). https://doi.org/10.17487/RFC2475

[2] George F. Riley and Thomas R. Henderson. 2010. The ns-3 Network Simulator. In *Modeling and Tools for Network Simulation*, Klaus Wehrle, Mesut Güneş, and James Gross (Eds.). Springer, Berlin, Heidelberg, 15–34. https://doi.org/10.1007/978-3-642-12331-3_2