A Network Simulator Differentiated Services Implementation

Open IP, Nortel Networks

Peter Pieda
<ppieda@nortelnetworks.com>
Jeremy Ethridge
<jethridg@nortelnetworks.com>
Mandeep Baines
Farhan Shallwani

July 26, 2000

TABLE OF CONTENTS

Abstract

This report details the implementation of the Differentiated Services (Diffserv) [1] architecture in the *ns* network simulator [2]. It provides an overview of ns, a brief explanation of Diffserv, and a thorough examination of the modules added to create an ns Diffserv implementation. The report explains both the C++ code that adds the Diffserv architecture and the Tcl scripts that access the Diffserv functionality.

1 Introduction

This document covers work performed at the Nortel Networks by Farhan Shallwani in the fall of 1998, Jeremy Ethridge in the summer of 1999, Peter Pieda in the Fall of 1999, and Mandeep Baines in the Spring of 2000. This paper is an update of Farhan Shallwani's report of December 1998, "Adding Support for the Differentiated Services IP QoS Architecture."

ns is a free network simulation program that can be downloaded from the web [3] and is compatible with a number of operating systems. The tool has substantial functionality for simulating different network topologies and traffic models. ns also has an open architecture that allows users to add new functionality.

Since the simulator lacks support for modeling Diffserv networks, new code was written to accommodate that architecture. This report concerns the creation, inner workings, and use of the new ns Diffserv modules.

2 Network Simulator Overview

2.1 Introduction

ns has been developed at the Lawrence Berkeley National Laboratory (LBNL) of the University of California, Berkeley (UCB). The extensibility of ns makes the tool very dynamic; changes occur frequently enough that a "daily snapshot" is available. Compatibility is maintained, however; so the Diffserv modules that were developed using ns-2.1b3 should work on any update of ns-2, which is the current version of ns.

ns is an event-driven network simulator. It has an extensible background engine implemented in C++ that uses OTcl (an object oriented version of Tcl [4]) as the command and configuration interface. Thus, the entire software hierarchy is written in C++, with OTcl used as a front end.

2.2 Tcl Scripts

A simulation is defined by an OTcl script. Running a simulation involves creating and executing a file with a ".tcl" extension, such as "simfile.tcl."

A Tcl ns script:

- Defines a network topology (including the nodes, links, and scheduling and routing algorithms of a network).
- Defines a traffic pattern (for example, the start and stop time of an FTP session).
- Collects statistics and outputs the results of the simulation. Results are usually written to files, including files for Nam [5], the Network Animator program that comes with the full ns download.

ns is an event-driven simulator that derives its functionality through an OTcl interpreter, which runs in the background. This interpreter translates each statement in the Tcl file into an event. For example, the statement:

```
$ns at 0.5 "$tcp start"
```

is translated into event, which at 0.5 seconds into the simulation, starts up a TCP source.

Marc Greis's ns tutorial [6] and the Tcl files already present in the "ns-2" directory present good examples on how to use the Tcl syntax correctly.

2.3 Software Architecture

ns is an object oriented simulator written in C++. This code serves as a backbone for the whole simulation process. The entire class hierarchy is implemented through this code; and the classes provide a wide array of network features. New classes or modules can be added by extending the current class hierarchy. An online documentation of the ns class hierarchy is maintained at [7].

Each class consists of the following of the following components:

Configuration parameters:

> Configuration parameters are class attributes that can be set and queried dynamically through the Tcl scripts. These simulation parameters are usually constant during the entire simulation, but they can be changed dynamically as desired. For example, the bandwidth of a link is usually set only at the start of a simulation. On the other hand, a traffic source can be configured to transmit packets of different sizes at different times.

State variables:

> Each class has a set of variables, many of which may be queried in a Tcl script to determine the state of that object. Usually, they are modified explicitly only when the object needs to be reset for another simulation run. For example, the length of a packet queue changes over time; and the instantaneous size of that queue can be queried through a Tcl command or used by a C++ method.

Methods:

> The methods associated with each class specify the actions that can be performed by that object. For example the *enque()* method for the RED gateway class specifies the enqueuing method for that object.

2.4 Adding a New Module

This section outlines the process of creating and adding new classes to the ns software hierarchy. The base Diffserv class detailed in Section 3.2 is used as an example. A closer look at the Diffserv architecture is provided in Section 2; but it is not necessary to understand the details of that architecture to follow this walkthrough. [6] and [2] also explain how to add a module to ns.

Adding a new module to ns consists of three steps:

Determining the need:

The Diffserv functionality is captured in a Queue object; it is an alternative to other queue types such as DropTail, CBQ, and RED. A Diffserv queue requires:

- The ability to implement multiple physical RED queues along a single link.
- The ability to implement multiple virtual queues on each physical queue, with independent sets of parameters for each virtual queue.
- The ability to determine in which physical and virtual queue a packet is enqueued, depending upon user specifications.

Determining the class hierarchy positioning:

Since this new class implements the generic Queue functionality and extends it with new methods and attributes, it is placed beneath Queue in the object hierarchy, as shown in the following figure:

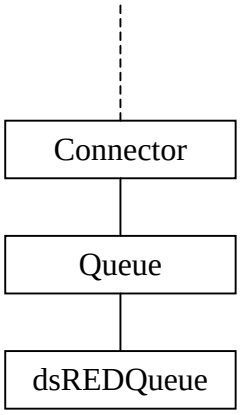

Figure 1.1 Position of dsREDQueue in the class hierarchy

Writing code:

Writing the code for the new module requires three or four steps, depending on the class:

Step 1: Creating the header file ("dsred.h")

This file includes class specifications, as well as other definitions needed by the new class.

Step 2: Creating the main C++ file ("dsred.cc")

This file includes implementations of each of the new class's methods. To incorporate the new class into ns and make it accessible through Tcl scripts, the class must be linked to the ns class hierarchy. The following code is used in "dsred.cc" to add *dsREDClass* to the class hierarchy:

```
static class dsREDClass : public TclClass {
public:
     dsREDClass() : TclClass("Queue/dsRED") {}
     TclObject* create(int, const char*const*) {
          return (new dsREDQueue);
     }
} class_dsred;
```

Step 3: Modifying "Makefile"

The third step is to add a reference to the new module in "Makefile," so that when the *make* command is invoked the compiler generates a binary version of the new code and includes it in the ns compilation. The reference is added to the *object files* section of "Makefile":

```
dsred.o
```

Step 4: Specifying default parameters for bound variables

A fourth step is necessary if configuration parameters are *bound* in the class constructor method. In that case, default values for all bound parameters should be added to the file "/ns-2/tcl/lib/ns-default.tcl." For example, the dsREDQueue constructor contains the binding:

```
bind("numQueues_", &numQueues_);
```

and the parameter is assigned default values in "/ns-2/tcl/lib/ns-default.tcl":

```
Queue/dsRED set numQueues_ 4
```

After completing those steps and recompiling ns with the *make* command, Tcl scripts can use the new class.

3 Differentiated Services Architecture for IP QoS

3.1 Introduction

Traditional IP networks offer users best-effort service. In this model, all user packets compete equally for network resources. This model has been sufficient until recently, when the usage and popularity of IP networks has soared. This rise has placed a significant burden on limited network resources, such as bandwidth and buffer space, resulting in heavy congestion. Such congestion does not encourage mass adoption of IP networks as transport mechanisms for real-time and mission-critical applications. Much attention is being given to developing IP Quality of Service (QoS), which allows network operators to offer differing levels of treatment to user packets.

Differentiated Services [1], or Diffserv, is an IP QoS architecture based on packet-marking that allows packets to be prioritized according to user requirements. A scheme known as Assured Forwarding [8] has been proposed as a potential user of Diffserv. Assured Forwarding provides differential treatment of traffic by discarding more low priority packets during times of congestion than high priority packets. Although the Assured Forwarding mechanism does not explicitly require a particular queue type, it is suited for RED [9].

3.2 An Overview of RED

RED (Random Early Detection) is a congestion avoidance algorithm that can be implemented in routers. The basic queue algorithm for routers is known as Drop Tail. Drop Tail queues simply accept any packet that arrives when there is sufficient buffer space and drop any packet that arrives when there is insufficient buffer space.

RED gateways instead attempt to detect incipit congestion by computing a weighted average queue size, since a sustained long queue is a sign of network congestion. Upon packet arrival, a RED gateway checks the weighted average queue size against specified minimum and maximum thresholds. If there is congestion, it notifies, either by dropping a packet or by setting a bit in a header field of the packet, probabilistically.

For a RED gateway that drops packets, rather than marking a congestion bit, the following three phases sum up its algorithm:

Phase1: Normal Operation

> If the average queue size is less than the minimum threshold, no packets are dropped.

Phase2: Congestion Avoidance

If the average queue size is between the minimum and maximum thresholds, packets are dropped with a certain probability. This probability is a function of the average queue size, so that larger queues lead to higher drop probabilities.

Phase3: Congestion Control

If the average queue size is greater than the maximum threshold, all incoming packets are dropped.

3.3 Multiple RED Parameters

The Diffserv architecture provides QoS by dividing traffic into different categories, marking each packet with a *code point* that indicates its category, and scheduling packets according to their code points. The Assured Forwarding mechanism is a group of code points that can be used in a Diffserv network to define four classes of traffic, each of which has three drop precedences. Those drop precedences enable differential treatment of traffic within a single class.

Assured Forwarding uses the RED mechanism by enqueuing all packets for a single class into one physical queue that is made up of three virtual queues (one for each drop precedence). Different RED parameters are used for the virtual queues, causing packets from one virtual queue to be dropped more frequently than packets from another. A packet with a lower drop precedence is given better treatment in times of congestion because it is assigned a code point that corresponds to a virtual queue with relatively lenient RED parameters.

For example, one code point might be used for assured traffic and another for best effort traffic. The assured packet virtual queue will have higher minimum and maximum thresholds than those of best effort queue, meaning that best effort packets will enter the congestion avoidance and congestion control phase prior to assured packets.

3.4 Diffserv Architecture

The Diffserv architecture has three major components. One is the policy and resource manager, which handles the creation of network policies and distribution of those policies to the Diffserv routers. The other components are edge routers and core routers. Diffserv attempts to restrict complexity to only the edge routers of a domain.

A policy specifies which traffic receives a particular level of service in the network. Although a policy and resource manager is a necessary component of a Diffserv network that allows an administrator to communicate policies to the edge and core devices, it is unimportant for the ns Diffserv implementation. Instead, policy information is simply specified for each edge and core device through the Tcl scripts.

The notion of edge and core devices, as illustrated in Figure 2.1, is key to the understanding of the ns Diffserv implementation, however.

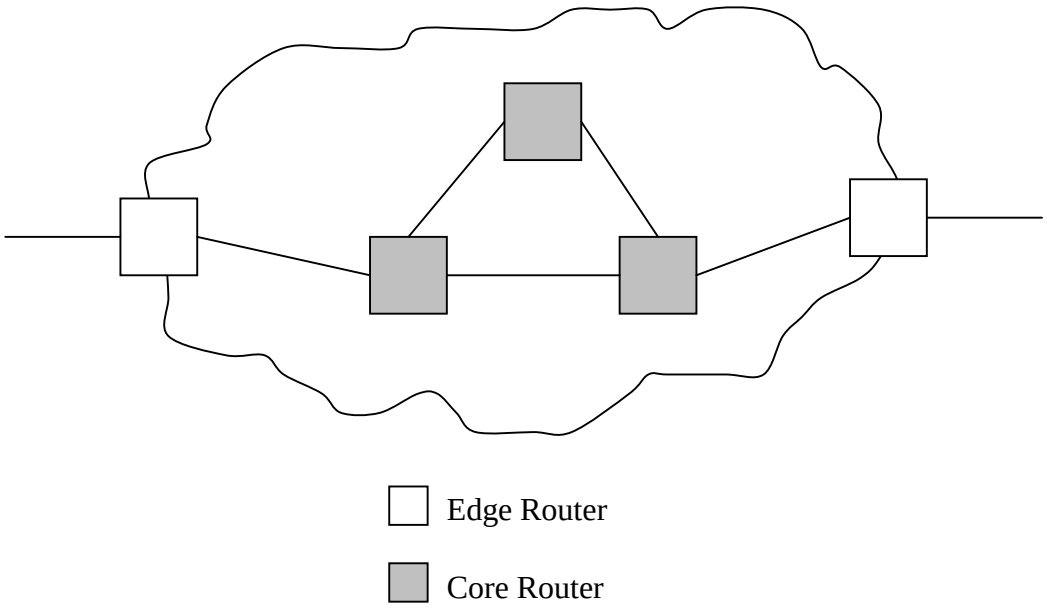

Figure 2.1 Devices in a Diffserv Domain

Edge Router Responsibilities:

- Examining incoming packets and classifying them according to policy specified by the network administrator.
- Marking packets with a code point that reflects the desired level of service.
- Ensuring that user traffic adheres to its policy specifications, by shaping and policing traffic.

Core Router Responsibilities:

- Examining incoming packets for the code point marking done on the packet by the edge routers.
- Forwarding incoming packets according to their markings. (Core routers provide a reaction to the marking done by edge routers.)

4 Diffserv Architecture ns Modules

4.1 Introduction

In order to design and implement the Diffserv architecture in ns, five modules had to be added to the class hierarchy: one for the base Diffserv router functionality (dsRED), one each for the edge and core routers, one for RED-based queuing and one for policing. Each module defines a single class.

4.2 dsRED Module

The dsRED module is the base module for the Diffserv implementation. It defines the *dsREDQueue* class, which is the parent class for the *edgeQueue* and *coreQueue* classes.

The dsRED module is contained in the files “dsred.h” and “dsred.cc.”

Purpose:

The dsREDQueue class is the parent class for the edgeQueue and coreQueue classes. It implements all functionality and declares all parameters that are common to edge and core Diffserv routers.

Class hierarchy positioning:

The dsREDQueue class extends the Queue class, as illustrated in the following figure.

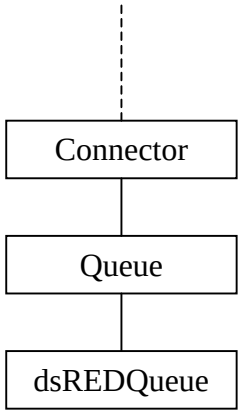

Figure 3.2 Position of dsREDQueue in the class hierarchy

Graphical representation:

A dsREDQueue uses the redQueue class, to form the following queue structure:

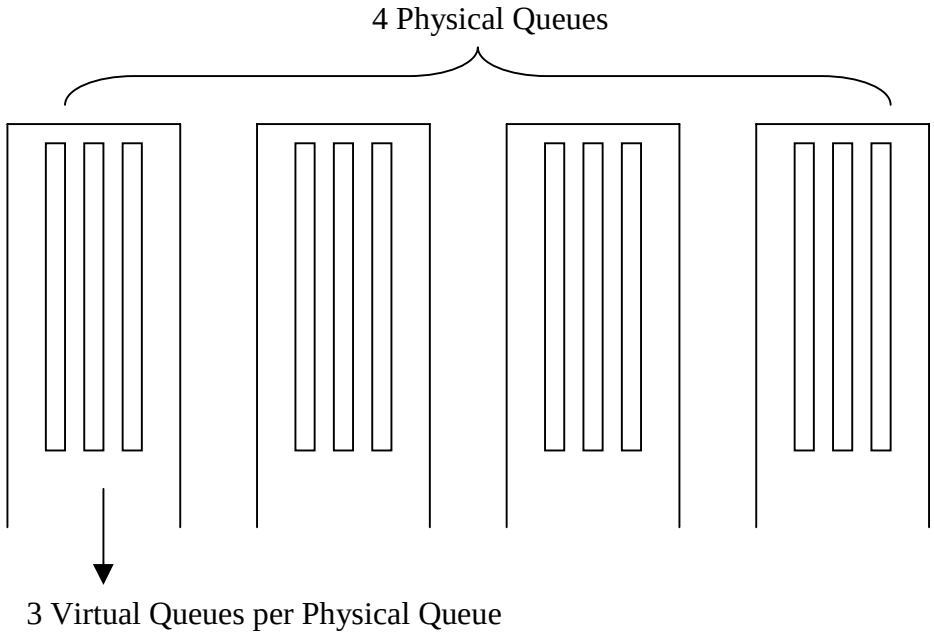

Figure 3.3 A dsREDQueue Instance

The queue structure consists of four physical queues, each containing three virtual RED queues, referred to as precedence levels. Each physical queue corresponds to a class of traffic; and each combination of a queue and precedence number is associated with a code point (or a drop preference).

Packets are enqueued in a certain queue and precedence number according to their code point marking. They are treated according to the corresponding parameters for that queue and precedence number; thus, a code point specifies a certain level of service.

The choice of four physical queues, each of which contains three virtual queues, is taken from [8]. Not all physical and virtual queues need be used; the user may configure a dsRED instance to have fewer physical or virtual queues through the Tcl script. These values may not be exceeded, however, without first altering the constants defined in “dsred.h” and recompiling ns.

PHB Table:

The dsREDQueue class contains a data structure known as the PHB Table (per hop behaviour table). Edge devices handle marking packets with code points and core devices simply respond to existing code points. However, both devices need to determine how to map a code point to a particular queue and precedence level.

The PHB Table handles this mapping by defining an array with three fields:

- Code point
- Class (Physical Queue)
- Precedence (Virtual Queue)

Tcl Configuration:

The configuration commands listed in this section apply to edgeQueue and coreQueue instances, since both classes are children of dsREDQueue.

Router configuration must be handled in the simulation before the arrival of any traffic.

The dsREDQueue class one bound variable: *numQueues_*.

The default value for numQueues_ is set in the file "/ns-2/tcl/lib/ns-default.tcl" as 4. This value can be changed inside Tcl scripts as follows, assuming that *dsredq* is a Tcl variable referring to a dsREDQueue (or child) object:

```
$dsredq set numQueues_ 1
```

numQueues_ refers to the number of physical queues.

The variant of MRED used to calculate queue sizes can be configured with the command:

```
$dsredq setMREDMode RIO-C 0
```

The above command sets the MRED mode of queue 0 to RIO-C. If the second argument was not included, all queues would be set to RIO-C. By default the MRED mode is set to RIO-C. The various MRED modes available are:

RIO-C (RIO Coupled): The probability of dropping an out-of-profile packet is based on the weighted average lengths of all virtual queues; while the probability of dropping an in-profile packet is based solely on the weighted average length of its virtual queue.

RIO-D (RIO De-coupled): The probability of dropping an out-of-profile packet is based on the size of its virtual queue.
WRED (WRED): All probabilities are based on a single queue length. This notion is explained in Section 3.2.2 of [10].
DROP (Weighted RED): As soon the queue size reaches the minimum threshold, all packets are dropped regardless of marking.

The number of virtual queues is not a bound variable, but can be configured with the command:

```
$dsredq setNumPrec 2
```

The numbers of physical and virtual queues are limited by constants inside the file "dsred.h," which should not be exceeded. No error checking is performed on the numQueues_ variable; it is assumed that the user will not exceed 4 physical queues. In general, only limited error-checking is performed on the Tcl configuration commands.

```
$dsredq configQ 0 1 10 20 0.10
```

This command configures the RED parameters for one virtual queue. The above example specifies that physical queue 0/virtual queue 1 has a min_{th} value of 10 packets, a max_{th} value of 20 packets, and a max_p value of 0.10. [9] contains an explanation of these RED parameters. For DropTail queues, only the first three parameters are required and the second is disregarded because there is no notion of precedence level for a DropTail queue.

```
$dsredq addPHBEntry 11 0 1
```

The addPHBEntry command adds an entry to the PHB Table; in this example, code point 11 is mapped to physical queue 0 and virtual queue 1. In ns, the packets are defaulted to a code point of zero. Therefore, to handle best effort traffic one must add a PHB entry for the zero code point.

```
$dsredq meanPktSize 1500
```

This command specifies the mean packet size (in bytes), which is used for RED calculations.

In addition, commands are available which allow the user to choose the scheduling mode between queues.

```
$dsredq setSchedularMode WRR
$dsredq addQueueWeights 1 5

The above pair of commands sets the scheduling mode to
Weighted Round Robin and then sets the queue weight
```

```
for queue 1 to 5. Other scheduling modes supported are
Weighted Interleaved Round Robin (WIRR), Round Robin
(RR), and Priority (PRI). The default scheduling mode
is Round Robin.

For Priority scheduling, priority is arranged in
sequential order with queue 0 having the highest
priority. Also, one can set the a limit on the maximum
bandwidth a particular queue can get using the
addQueueRate command.

$dsredq setSchedularMode PRI
$dsredq addQueueRate 0 5000000

For example, the above set of commands set the
scheduling mode to Priority and puts a limit on the
queue 0 bandwidth to 5 Mbps.
```

Tcl Querying:

The values of the bound variables may be checked from a script; and the dsREDQueue Tcl interface also interprets three additional queries:

```
$dsredq printPHBTable
```

This command prints the entire PHB Table, one line at a time.

```
$dsredq printStats
```

This command is meant to be a debugging tool that can be altered as needed. Currently, it prints the defined number of physical and virtual queues.

```
$dsredq getAverage 0
```

This query returns the RED weighted average size of the specified physical queue.

4.3 redQueue Class

redQueue class defines a physical RED queue composed of multiple virtual queues.

The dsRED module is contained in the files “dsredq.h” and “dsredq.cc.”

Graphical representation:

The redQueue class implements a single physical RED queue that contains multiple virtual queues, as illustrated in the following figure:

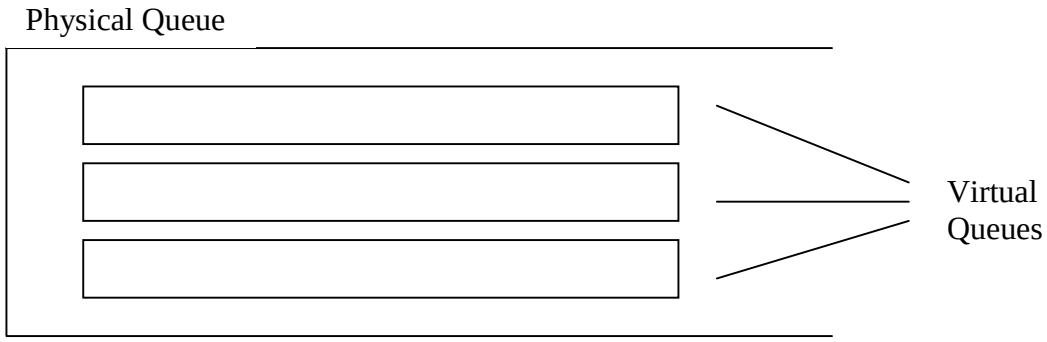

Figure 3.1 A redQueue Instance

One underlying physical queue incorporates multiple virtual RED queues, each of which has a distinct set of RED parameters.

Purpose:

The purpose of the Diffserv architecture is to provide different treatment to different traffic. The redQueue class enables that differentiation by defining virtual RED queues, each of which has independent configuration and state parameters.

For example, the length of each virtual queue is calculated only on packets mapped to that queue. Thus, packet dropping decisions can be applied based on the state and configuration parameters of the virtual queues.

The redQueue class is not equivalent to the REDQueue class, which was already present in ns. Instead, it is a modified copy of that class that includes the notion of virtual queues.

Tcl Configuration and Querying:

Instances of the redQueue class only exist inside instances of the dsREDQueue class. All user interaction with the redQueue class is handled through the command interface of the dsREDQueue class.

4.4 Edge Module

The edge module implements a Diffserv edge router. It defines the *edgeQueue* class, which models an edge router.

The edge module is contained in the files “edge.h” and “edge.cc.”

Purpose:

The edgeQueue class, as a child of the dsREDQueue class is responsible for maintaining multiple physical and virtual queues and processing those queues according to their parameters. Its additional responsibilities are:

- Marking packets with code points.
- Policing traffic aggregates.

Class Hierarchy positioning:

The edgeQueue class is an extension of the dsREDQueue class:

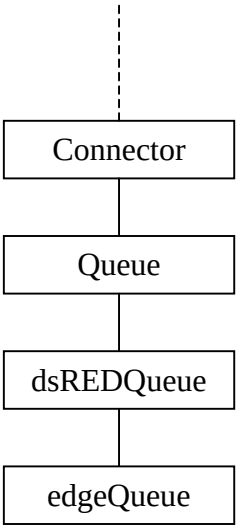

Figure 3.4 Position of edgeQueue in the class hierarchy

Policy:

The edgeQueue class contains an instance of the Policy class that handles all policy creation and enforcement. The Policy class is examined in Section 3.3.2.

Tcl Configuration and Querying:

Since most of the Tcl interface commands deal with policy data, an examination of edgeQueue's command interface is given in Section 3.3.3, after the Policy class is explained.

4.5 Policy Class

The *Policy* class is used by the edgeQueue class to handle all policy functionality.

The policy module is contained in the files "dsPolicy.h" and "dsPolicy.cc."

Purpose:

The Policy class handles the creation, manipulation, and enforcement of edge router policies. A policy determines the treatment that a traffic aggregate will receive at the edge device. Edge devices use policy information to determine with what code point to mark packets.

Policy Overview:

A policy is established between a source and destination node. All flows matching that source-destination pair are treated as a single *traffic aggregate*. Each policy defines a *policer type*, a *target rate*, and other policer-specific parameters. As a minimum, each policy defines two code points; and the choice of code point depends on a comparison between the aggregate's target rate and current sending rate.

General Mechanism:

Each traffic aggregate has an associated policer type, meter type, and initial code point. The meter type specifies the method for measuring the state variables needed by the policer. For example, the TSW Tagger [10] is a meter that measures the average traffic rate, using a specified time window.

When a packet arrives at an edge device, it is examined to determine to which aggregate it belongs. The meter specified for that aggregate is invoked to update all state variables. Then the policer is invoked to determine how to mark the packet. Depending on the aggregate's state variables, either the specified initial code point is used or a downgraded code point is used; and the packet is enqueued accordingly.

Policy Table:

The Policy class uses a Policy Table to store the policies of each traffic aggregate. This table is an array that includes fields for the source and destination nodes, a policer type, a meter type, an initial code point, and various state information. For

each policer type, only some of the state variables are used. The wasted memory space taken up by the unused fields is not considered significant.

The fields of the Policy Table are:

- Source node ID
- Destination node ID
- Policer type
- Meter type
- Initial code point
- CIR (committed information rate)
- CBS (committed burst size)
- C bucket (current size of the committed bucket)
- EBS (excess burst size)
- E bucket (current size of the excess bucket)
- PIR (peak information rate)
- PBS (peak burst size)
- P bucket (current size of the peak bucket)
- Arrival time of last packet
- Average sending rate
- TSW window length

Policer Types:

The Policy class currently supports six policer types.

TSW2CM (TSW2CMPolicer): uses a CIR and two drop precedences. The lower precedence is used probabilistically when the CIR is exceeded.

TSW3CM (TSW3CMPolicer): uses a CIR, a PIR, and three drop precedences. The medium drop precedence is used probabilistically when the CIR is exceeded and the lowest drop precedence is used probabilistically when the PIR is exceeded, as explained in [11].

Token Bucket (tokenBucketPolicer): uses a CIR and a CBS and two drop precedences. An arriving packet is marked with the lower precedence if and only if it is larger than the token bucket.

Single Rate Three Color Marker (srTCMPolicer): uses a CIR, CBS, and an EBS to choose from three drop precedences, as explained in [12].

Two Rate Three Color Marker (trTCMPolicer): uses a CIR, CBS, PIR, and a PBS to choose from three drop precdences, as explained in [13].

Meter Types:

The metering and policing methods are decoupled inside "edge.cc"; but currently each policer type maps to a specific policer. The two TSW policers all use the TSW Tagger described in [10]. Each of the other policer types has its own associated meter.

Policer Table:

Each policer takes an initial code point and chooses whether to retain it or downgrade it. The downgrading is consistent within a policer type. If two aggregates use the same policer and initial code point, each is downgraded to the same code point(s).

The Policy class uses a Policer Table to store the mappings from a policy type and initial code point pair to its associated downgraded code point(s). This table is an array with four fields:

- Policer type
- Initial code point
- Downgraded code point 1
- Downgraded code point 2

The last field is not used for policer types with only two drop precedences; and it should be set to the worst code point for policer types with three drop precedences.

Configuration:

The addPolicyEntry command is used to add an entry to the Policy Table. It takes different parameters depending on what policer type is used. The first two parameters after the command name are always the source and destination node IDs, and the next parameter is the policer type.

Following the policer type are the parameters needed by that policer as summarized below:

- TSW2CM Initial code point CIR
- TSW3CM Initial code point CIR PIR
- TokenBucket Initial code point CIR CBS
- srTCM Initial code point CIR CBS EBS
- trTCM Initial code point CIR CBS PIR PBS

The rates CIR and PIR are specified in bits per second; the buckets CBS, EBS, and PBS are specified in bytes.

Consider a Tcl script for which $q is a variable for an edge queue, and $s and $d are source and destination nodes. The following command adds a TSW2CM policer for traffic going from the source to the destination:

```
$q addPolicyEntry [$s id] [$d id] TSW2CM 10 2000000
```

The following parameters could be used in place of "TSW2CM . . ." to use a different policer:

```
TSW3CM 10 2000000 3000000
TokenBucket 10 2000000 10000
srTCM 10 2000000 10000 20000
trTCM 10 2000000 10000 3000000 10000
```

Note, however, that only one policy can be applied to any source-destination pair.

The following command adds an entry to the Policer Table, specifying that the trTCM initial (green) code point 10 should be downgraded to yellow code point 11 and red code point 12:

```
$dsredq addPolicerEntry trTCM 10 11 12
```

There must be a Policer Table entry in place for every policer type/initial code point pair.

Querying:

Four queries are interpreted by an edgeQueue class instance:

```
$dsredq printPolicyTable
```

This command prints the entire Policy Table, one line at a time.

```
$dsredq printPolicerTable
```

This command prints the entire Policer Table, one line at a time.

```
$dsredq getCBucket
```

This query returns the current size of the C Bucket, in bytes.

4.6 Core Module

Purpose:

This class emulates the core router in the Diffserv architecture; thus, is intended to work downstream from an edge router. It forwards packets according to the marking done on them by the edge router. Packets having code points signifying low priority are dropped at a considerably higher rate than packets marked with code points of high priority.

The core module is contained in the files “core.h” and “core.cc.”

Class hierarchy positioning:

This class inherits its behaviour from dsREDQueue class; therefore it is positioned as illustrated here:

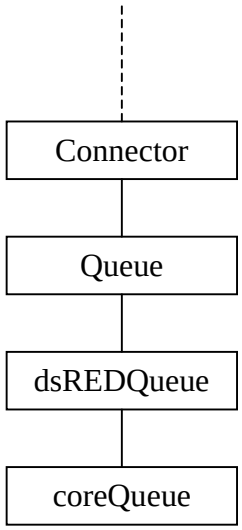

Figure 3.5 Position of coreQueue in the class hierarchy

5 A Look at Sample Tcl Scripts

5.1 A Simple Token Bucket Test

To summarize the Diffserv ns functionality, a simple script will be examined in detail. Most directories include a “README” file that explains the scripts or data files in that directory.

The script that will be examined is “cbr-tb.tcl.” The “policer_tests” directory contains simple example scripts for each of the policer types, as explained in the “README” file. “cbr-tb.tcl” uses CBR traffic sources and Token Bucket policers.

The entire script will now be examined, from beginning to end:

```
#---------------------------------------------------------------------------
# cbr-tb.tcl
# Author: Jeremy Ethridge.
# Dates: June 29-July 5, 1999.
# Notes: A DS-RED script that uses CBR traffic agents and the Token Bucket policer.
#
#    ----
#    |s1|-----------
#    ----   10 Mb   \
#            5 ms    \
#                     \----           ------          ----           ------
#                      |e1|-----------|core|----------|e2|-----------|dest|
#                     /----   10 Mb   ------   5 Mb   ----   10 Mb   ------
#                    /         5 ms            5 ms           5 ms
#    ----           /
#    |s2|-----------
#    ----   10 Mb
#            5 ms
#
#---------------------------------------------------------------------------
```

The introductory header comments describe the test and illustrate the network topology.

```
set ns [new Simulator]

set cir0  1000000
set cbs0     3000
set rate0 2000000
set cir1  1000000
set cbs1    10000
set rate1 3000000

set testTime 85.0
set packetSize 1000
```

This piece of code declares the Simulator instance and defines constants that are used later in the simulation.

```
# Set up the network topology shown at the top of this
file:
set s1 [$ns node]
set s2 [$ns node]
set e1 [$ns node]
set core [$ns node]
set e2 [$ns node]
set dest [$ns node]
```

These lines declare each of the nodes in the topology.

```
$ns duplex-link $s1 $e1 10Mb 5ms DropTail
$ns duplex-link $s2 $e1 10Mb 5ms DropTail

$ns simplex-link $e1 $core 10Mb 5ms dsRED/edge
$ns simplex-link $core $e1 10Mb 5ms dsRED/core

$ns simplex-link $core $e2 5Mb 5ms dsRED/core
$ns simplex-link $e2 $core 5Mb 5ms dsRED/edge

$ns duplex-link $e2 $dest 10Mb 5ms DropTail
```

Next, the links are created. *Notice that edge queues are used only along the links from edge queues into the network domain (from the two edge devices to the core) and that core queues are used only along links originating from the core router.*

```
set qE1C [[$ns link $e1 $core] queue]
set qE2C [[$ns link $e2 $core] queue]
set qCE1 [[$ns link $core $e1] queue]
set qCE2 [[$ns link $core $e2] queue]
```

This code obtains handles to each of the four Diffserv queues. These handles are necessary for configuring the Diffserv queues.

```
# Set DS RED parameters from Edge1 to Core:
$qE1C meanPktSize $packetSize
$qE1C set numQueues_ 1
$qE1C setNumPrec 2
$qE1C addPolicyEntry [$s1 id] [$dest id] TokenBucket 10
$cir0 $cbs0
$qE1C addPolicyEntry [$s2 id] [$dest id] TokenBucket 10
$cir1 $cbs1
$qE1C addPolicerEntry TokenBucket 10 11
$qE1C addPHBEntry 10 0 0
$qE1C addPHBEntry 11 0 1
$qE1C configQ 0 0 20 40 0.02
$qE1C configQ 0 1 10 20 0.10
```

This block of the script configures all of the parameters for the edge queue between nodes Edge1 and Core. The *meanPktSize* command is required for the RED state variables to be calculated accurately. Setting the number of queues and precedence levels is optional, but it aids efficiency. Because neither the scheduling or MRED mode type are set, they default to Round Robin scheduling and RIO-C Active Queue Management.

The *addPolicyEntry* commands establish two policies at the edge queue: one between nodes S1 and Dest and one between nodes S2 and Dest. Note that the *[$s1 id]* command returns the ID value needed by addPolicyEntry. The CIR and CBS values used in the policies are the ones set at the beginning of the script.

The *addPolicerEntry* line is required because each policer type/initial code point pair requires an entry in the Policer Table. Each of the policies uses the same policer and initial code point, so only one entry is needed.

The *addPHBEntry* commands map each code point to a queue/precedence pair. Although each code point in this example maps to a unique queue/precedence pair, that need not be the case; multiple code points could receive identical treatment.

Finally, the *configQ* commands set the RED parameters for each virtual queue. Note that as the precedence value increases, the RED parameters become harsher, which follows [8].

```
# Set DS RED parameters from Edge2 to Core:
$qE2C meanPktSize $packetSize
$qE2C set numQueues_ 1
$qE2C setNumPrec 2
$qE2C addPolicyEntry [$dest id] [$s1 id] TokenBucket 10
$cir0 $cbs0
$qE2C addPolicyEntry [$dest id] [$s2 id] TokenBucket 10
$cir1 $cbs1
$qE2C addPolicerEntry TokenBucket 10 11
$qE2C addPHBEntry 10 0 0
$qE2C addPHBEntry 11 0 1
$qE2C configQ 0 0 20 40 0.02
$qE2C configQ 0 1 10 20 0.10
```

The configuration of the second edge queue matches the first with one exception. The policies now apply from the destination node to the source nodes, since this edge device is an ingress point for the return data path.

It should also be noted that different policy types and parameters, code points, and queue mappings could have been used along this link. The only requirement of an edge queue is that it has a valid policy in place for all incoming traffic.

```
# Set DS RED parameters from Core to Edge1:
$qCE1 meanPktSize $packetSize
$qCE1 set numQueues_ 1
$qCE1 setNumPrec 2
$qCE1 addPHBEntry 10 0 0
$qCE1 addPHBEntry 11 0 1
$qCE1 configQ 0 0 20 40 0.02
$qCE1 configQ 0 1 10 20 0.10
```

Note that the configuration of a core queue matches that of an edge queue, except that there is no Policy Table or Policer Table to configure at a core router. A core router's chief requirement is that it has a PHB entry for all code points that it will see.

```
# Set DS RED parameters from Core to Edge2:
$qCE2 meanPktSize $packetSize
$qCE2 set numQueues_ 1
$qCE2 setNumPrec 2
$qCE2 addPHBEntry 10 0 0
$qCE2 addPHBEntry 11 0 1
$qCE2 configQ 0 0 20 40 0.02
$qCE2 configQ 0 1 10 20 0.10
```

The other core queue is configured the same. As with the edge queues, there is no requirement that the core queue parameters be the same.

```
# Set up one CBR connection between each source and the
destination:
set cbr0 [new Agent/CBR]
$ns attach-agent $s1 $cbr0
$cbr0 set packetSize_ $packetSize
$cbr0 set interval_ [expr 1.0 / [expr $rate0 / 8000.0]]
set null0 [new Agent/Null]
$ns attach-agent $dest $null0
$ns connect $cbr0 $null0

set cbr1 [new Agent/CBR]
$ns attach-agent $s2 $cbr1
$cbr1 set packetSize_ $packetSize
$cbr1 set interval_ [expr 1.0 / [expr $rate1 / 8000.0]]
set null1 [new Agent/Null]
$ns attach-agent $dest $null1
$ns connect $cbr1 $null1
```

This code is not related to the Diffserv modules. It establishes a CBR connection between each source node and the destination node.

```
proc finish {} {
    global ns
    exit 0
}
```

This is simply a terminating procedure.

```
$qE1C printPolicyTable
$qE1C printPolicerTable

$ns at 0.0 "$cbr0 start"
#$ns at 0.0 "$cbr1 start"
$ns at 20.0 "$qCE2 printCoreStats"
$ns at 40.0 "$qCE2 printCoreStats"
$ns at 60.0 "$qCE2 printCoreStats"
$ns at 80.0 "$qCE2 printCoreStats"
$ns at $testTime "$cbr0 stop"
#$ns at $testTime "$cbr1 stop"
$ns at [expr $testTime + 1.0] "finish"

$ns run
```

Finally, these lines start and stop the simulation, and output different statistics. Note that one of the CBR sources is commented out, so that it does not start or stop. The output of this script is shown below:

```
Policy Table(2):
Flow (0 to 5): Token Bucket policer, initial code  point
                    10, CIR 1000000.0 bps, CBS 3000.0
                    bytes.
Flow (1 to 5): Token Bucket policer, initial code  point
                    10, CIR 1000000.0 bps, CBS 10000.0
                    bytes.

Policer Table:
Token Bucket policer code point 10 is policed to code point
                    11.


Packets Statistics
=======================================
 CP  TotPkts   TxPkts   ldrops   edrops
 --  -------   ------   ------   ------
All    12494    12494        0        0
 10     5009     5009        0        0
 11     7485     7485        0        0

Packets Statistics
=======================================
 CP  TotPkts   TxPkts   ldrops   edrops
 --  -------   ------   ------   ------
All    24994    24994        0        0
 10    10009    10009        0        0
 11    14985    14985        0        0
```

```
Packets Statistics
=====================================
 CP  TotPkts   TxPkts   ldrops   edrops
 --  -------   ------   ------   ------
All    37494    37494        0        0
 10    15009    15009        0        0
 11    22485    22485        0        0

Packets Statistics
=====================================
 CP  TotPkts   TxPkts   ldrops   edrops
 --  -------   ------   ------   ------
All    49994    49994        0        0
 10    20009    20009        0        0
 11    29985    29985        0        0
```

References

[1] Blake, S., D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. *"An Architecture for Differentiated Services."* RFC 2475. December 1998.

[2] Fall, K. and K. Varadhan, editors. *"ns Notes and Documentation."* http://www.isi.edu/nsnam/ns/ns-documentation.html

[3] http://www.isi.edu/nsnam/ns/

[4] http://dev.ajubasolutions.com/

[5] http://www.isi.edu/nsnam/nam/

[6] Greis, Marc. *"Tutorial for the Network simulator 'ns.'"* http://www.isi.edu/nsnam/ns/tutorial/index.html

[7] http://www-sop.inria.fr/rodeo/Antoine.Clerget/ns/

[8] Heinanen, J., F. Baker, W. Weiss, and J. Wroclawski. *"Assured Forwarding PHB Group."* RFC 2597. June 1999.

[9] Floyd, S. and V. Jacobson. *"Random Early Detection Gateways for Congestion Avoidance."* 1993.

[10] Clark, D. and W. Fang. *"Explicit Allocation of Best Effort Packet Delivery Service."*

[11] Fang, W., N. Seddigh, and B. Nandy. *"A Time Sliding Window Three Color Marker"* (Internet draft). March, 2000.

[12] Heinanen, J., T. Finland, and R. Guerin. *"A Single Rate Three Color Marker"* (Internet draft). May, 1999.

[13] Heinanen, J., T. Finland, and R. Guerin. *"A Two Rate Three Color Marker"* (Internet draft). May, 1999.