The ns-3 DOCSIS® Module Documentation

Release v0.1

CableLabs

Jun 03, 2020

CONTENTS

This is the stand-alone version of the ns-3 DOCSIS module documentation.

Disclaimer

CHAPTER

ONE

INTRODUCTION

The DOCSIS® extension module for ns-3 (`docsis-ns3`) allows users to experiment with models of low latency DOCSIS® operation in the ns-3 simulation environment.

1.1 Who Should Use This Guide

This guide is intended for researchers who are interested in developing packet level simulations of networks that contain DOCSIS 3.1 cable broadband links. The guide gives a brief overview of *ns-3*, describes the necessary steps to build the module and run some example experiments, and it discusses some of the internal architecture and current limitations of the model.

1.2 Getting Started

1.2.1 What is ns-3?

ns-3 is an open-source packet-level network simulator. ns-3 is written in C++, with optional Python bindings. ns-3 is a command-line tool that uses native C++ as its modeling languange. Users must be comfortable with at least basic C++ and compiling code using g++ or clang++ compilers. Linux and MacOS are supported; Windows native Visual Studio C++ compiler is not supported, but Windows 10 machines can run ns-3 either through the Bash Subsystem for Linux, or on a virtual machine.

An ns-3 simulation program is a C++ main() executable, or a Python program, that links the necessary libraries and constructs a simulation scenario to generate output data. Users are often interested in conducting a study in which scenarios are re-run with slightly different configurations. This is usually accomplished by a script written in Bash or Python (or another scripting language) calling the ns-3 program with slightly different configurations, and taking care to label and save the output data for post-processing. Data presentation is usually done by users constructing their own custom scripts and generating plots through tools such as Matplotlib or gnuplot.

Some animators, visualizers, and graphical configuration editors exist for ns-3 but are not actively maintained.

1.2.2 ns-3 documentation

A large amount of documentation on *ns-3* is available at https://www.nsnam.org/documentation. New readers are suggested to thoroughly read the *ns-3* tutorial.

Please note that this documentation attempts to quickly summarize how users can get started with the specific features related to DOCSIS. There are portions of ns-3 that are not relevant to DOCSIS simulations (e.g. the Python bindings or NetAnim network animator) so we will skip over them.

1.2.3 What version of ns-3 is this?

This extension module is designed to be run with ns-3.31 release (expected for May 2020) or later versions of ns-3.

1.2.4 Prerequisites

This version of ns-3 requires, at minimum, a modern C++ compiler supporting C++11 (g++ or clang++), a Python 3 installation, and Linux or macOS.

For Linux, distributions such as Ubuntu 16.04, RedHat 7, or anything newer, should suffice. For macOS, users will either need to install the Xcode command line tools or the full Xcode environment.

We have added experimental control and plotting scripts that have additional Python dependencies, including:

- `matplotlib`: Consult the Matplotlib installation guide: https://matplotlib.org/faq/installing_faq.html.
- `reportlab`: Typically, either `pip install reportlab` or `easy_install reportlab`
- *pillow*: The Python Imaging Library (now maintained as pillow). Typically, `pip install pillow` or `easy_install pillow`
- A PDF concatenation program, either “`PDFconcat`”, “`pdftk`”, or “`pdfunite`”

For Mac users: `PDFconcat` is simply an alias to `/System/Library/Automator/Combine PDF Pages.action/Contents/Resources/join.py`

1.2.5 What is *waf*?

This is a Python-based build system, similar to `make`. See the ns-3 documentation (https://www.nsnam.org/documentation) for more information.

1.2.6 How do I build ns-3?

There are two steps, `waf configure` and `waf build`.

There are two main build modes supported by waf: *debug* and *optimized*. When running a simulation campaign, use *optimized* for faster code. If you are debugging and want ns-3 logging, use *debug* code.

Try this set of commands to get started from within the top level ns-3 directory:

```
$ ./waf configure -d optimized --enable-examples --enable-tests
$ ./waf build
$ ./test.py
```

The last line above will run all of the ns-3 unit tests. To build a debug version:

```
$ ./waf configure -d debug --enable-examples --enable-tests
$ ./waf build
```

1.2.7 `waf configure` reports missing features?

You will see a configuration report after typing `./waf configure` that looks something like this:

```
---- Summary of optional NS-3 features:
Build profile                 : optimized
Build directory               :
BRITE Integration             : not enabled (BRITE not enabled (see option --with-
↪brite))
DES Metrics event collection  : not enabled (defaults to disabled)
Emulation FdNetDevice         : enabled
...
```

Do not worry about the items labeled as *not enabled*; you will not need them for DOCSIS simulations.

1.2.8 Where are the interesting programs located?

The `examples/` directory contains an example DOCSIS simulation program. Only one example, `residential-example.cc`, is provided at this time.

the `experiments/residential/` contains plotting and execution scripting around this program, to automate the running of some interesting experiments.

Try these commands:

```
$ cd experiments/residential
$ ./residential.sh test
```

After the build information is displayed (showing what modules are enabled and disabled), you should see something like this, indicating a number of processes have been spawned in parallel in the background:

```
****************************************************************
* Launched:  results/test-20200220-190624/residential.sh
* Output in:  results/test-20200220-190624/commandlog.out
* Kill this run with:  kill -SIGTERM -30307
****************************************************************
```

When all simulations have finished, you can recurse into the timestamped directory named: `results/test-YYYYMMDD-HHMMSS` to find the outputs.

More thorough documentation about this is found in the same directory (in Markdown format) in the file named `residential-documentation.md`.

Users can also inspect the unit test programs in `test/` for simpler examples of how to put together simulations (although the test code is constructed for testing purposes).

1.2.9 Editing the code

In most cases, the act of running a program or experiment script will trigger the rebuilding of the simulator if needed, but you can force a rebuild by typing `./waf build` at the top-level ns-3 directory.

CHAPTER

TWO

MODEL OVERVIEW

The Data Over Cable Service Interface Specification (DOCSIS) specifications [DOCSIS3.1] are used by vendors to build interoperable equipment comprising two device types: the cable modem (CM) and the cable modem termination system (CMTS). DOCSIS links are multiple-access links in which access to the uplink and downlink channels on a hybrid fiber/coax (HFC) plant is controlled by a scheduler at the CMTS.

This module contains *ns-3* models for sending Internet traffic between CM and CMTS over an abstracted physical layer channel model representing the HFC plant. These *ns-3* models are focused on the DOCSIS MAC layer specification for low-latency DOCSIS version 3.1, version I19, Oct. 2019 [DOCSIS3.1.I19].

The *ns-3* models contain high-fidelity models of the MAC layer packet forwarding operation of these links, including detailed models of the active queue management (AQM) and MAC-layer scheduling and framing. Other aspects of MAC layer operation are highly abstracted. For example, no MAC Management Messages are exchanged between the CM and CMTS model.

In brief, these models focus on the management of packet forwarding in a downstream (CMTS to a single cable modem) or upstream (single cable modem to CMTS) direction, by modeling the channel access mechanism (requests and grants), scheduling, and queueing (via Active Queue Management (AQM)) present in the cable modem and CMTS.

The physical channel is a highly abstracted model ofthe Orthogonal Frequency Division Multiplexing (OFDM)-based downstream and OFDM with Multiple Access (OFDMA)-based upstream PHY layer. The channel model supports all of the basic DOCSIS OFDM/OFDMA PHY configuration options (#subcarriers, bit loading, frame sizes, interleavers, etc.), and can model physical plant length. There are no physical-layer impairments implemented.

Channel bonding is not currently supported.

Each instance of model supports a single CM / CMTS pair, and supports either a single US/DS Service Flow pair (with or without PIE AQM), or an LLD single US/DS Low Latency Aggregate Service Flow pair (each with a Classic SF and Low Latency SF). Upstream shared-channel congestion (i.e. multiple CMs contending for access to the channel) can be supported via an abstracted congestion model, which supports a time-varying load on the channel. There is currently no downstream shared-channel congestion model.

The model supports both contention requests and piggyback requests, but the contention request model is simplified - there are no request collisions, the CM always succeeds in sending a request sometime (selected via uniform random variable) in the next MAP interval. The scheduling types supported are Best Effort and PGS. Supported QoS Params are: Max Sustained Rate, Peak Rate, Max Burst, Guaranteed Grant Rate. The PGS scheduler supports only two values for Guaranteed Grant Interval: GGI = MAP interval & GGI = OFDMA Frame interval. It also implements the LLD queuing functions, including dual-queue-coupled-AQM and queue-protection.

This model started as a port of an earlier *ns-2* model of DOCSIS 3.0, to which extensions to model OFDMA framing were added. Some original authors of the DOCSIS 3.0 *ns-2* model are credited in these models.

2.1 Model Description

This model is organized into a single `docsis-ns3` *ns-3* extension module, which can be placed either in the *ns-3* `src` or `contrib` directory. Typically, extension modules are placed in the `contrib` directory.

There are two specialized `NetDevice` classes to model the DOCSIS downstream and upstream devices, and a special Channel class to interconnect them. The base class `DocsisNetDevice` supports both the `CmNetDevice` and the `CmtsNetDevice` classes.

The main modeling emphasis of these DOCSIS NetDevice models is to model the latency due to scheduling, queueing, fragmentation, and framing at the MAC layer. Consequently, there is a lot of abstraction at the physical layer.

2.1.1 DocsisNetDevice

As mentioned above, the `DocsisNetDevice` class is the base class for `CmNetDevice` and `CmtsNetDevice`. Its attributes and methods include things that are in common between the CM and CMTS, such as upstream and downstream channel parameters, MAC-layer framing, etc. The salient attributes are described in a later section of this document.

2.1.2 CmtsNetDevice

The CMTS functionality is encapsulated in an ns-3 class `CmtsNetDevice` modeling a DOCSIS downstream link (from CMTS to the cable modem). More specifically, it models a single OFDM downstream channel upon which either a single downstream service flow or a single downstream aggregate service flow (with underlying "classic" and "low-latency" service flows) is instantiated to provide service to a single cable modem.

As per the DOCSIS 3.1 specification, each service flow uses two token buckets (peak and sustained rate) for rate shaping that will accumulate tokens according to their parameters.

2.1.3 CmNetDevice and CmtsUpstreamScheduler

`CmNetDevice` device type models a DOCSIS upstream link (from cable modem to the CMTS). More specifically, it models a single upstream OFDMA channel upon which either a single upstream service flow or a single upstream aggregate service flow (with underlying "classic" and "low-latency" service flows) is instantiated to provide service to a single cable modem.

DOCSIS's upstream transmission is scheduled in regular intervals called "MAP intervals" (typically 1 - 2ms). Before the beginning of each MAP interval, the cable modem receives a MAP message potentially providing grants for its service flows during the MAP interval. The size of the grant will depend on the backlog of bytes requested by the CM, the state of the rate shaper, and congestion from other users on the shared upstream link (see below).

The model is based upon generating events corresponding to the MAP interval and a notional request-grant interaction between the `CmNetDevice` and an instance of the `CmtsUpstreamScheduler` object. While the `CmtsUpstreamScheduler` is notionally implemented at the CMTS, in the model there is no direct linkage between the `CmtsUpstreamScheduler` object and the CmtsNetDevice.

The following events periodically occur every MAP interval:

1. At a particular time in advance of the start of the MAP interval, the `CmtsUpstreamScheduler` generates the MAP schedule by calculating how many "minislots" that it will grant to each service flow and when within the MAP interval each grant is scheduled. The CMTS then notionally transmits this information in a MAP message to the CM in advance of the MAP interval by scheduling an event corresponding to its arrival at the CM.

2. At MAP message arrival, the CM parses the MAP message and creates an event shortly before each grant time. If there is no grant for a service flow within the MAP interval, a contention request opportunity is identified at MAP arrival time, and an event is scheduled.
3. At each grant event, the CM dequeues packets, calculates any new requests (for piggybacking), notionally builds the L2 transmission frame, and schedules events corresponding to packet arrivals at the CMTS.
4. At each contention request event, the CM calculates any new requests for the service flow, and schedules an event corresponding to the arrival of the request frame at the CMTS.

A key aspect of the model is that there is no actual exchange of *ns-3* Packet objects between the CM and CMTS for the control plane; instead, a MAP message reception is *simulated* at the CM as if the CMTS had sent it, and Request message reception is *simulated* at the CMTS as if the CM had sent it. The class `CmtsUpstreamScheduler` is responsible for the scheduling.

The `CmtsUpstreamScheduler` also has an abstract congestion model that can be used to induce scheduling behavior corresponding to shared channel congestion (in which not all requested bytes can be granted in the next MAP interval). This is described in the next subsection.

DOCSIS 3.1 upstream frame transmission utilizes "continuous concatenation and fragmentation" in which L2 DOCSIS MAC frames (an Ethernet frame prepended with a DOCSIS MAC Frame Header) are enqueued as a contiguous string of bytes, and then to fill a grant, an appropriate number of bytes are dequeued (without regard to DOCSIS MAC frame boundaries). The resulting set of bytes has a Segment Header prepended before transmission. As a result, each upstream transmission (i.e. each grant) could contain the tail end of one MAC frame, some number of whole MAC frames, and the head of another MAC frame. In the *ns-3* model this is abstracted by simply calculating when the tail of each MAC frame would be transmitted, and then scheduling the arrival of the corresponding Packet object at the CMTS at the notional time when the tail would have arrived.

The upstream channel in DOCSIS is composed of OFDMA frames, with a configured duration. Each frame consists of a set of minislots, each with a certain byte capacity. All frames have the same number of minislots. The minislots are numbered serially as shown below.

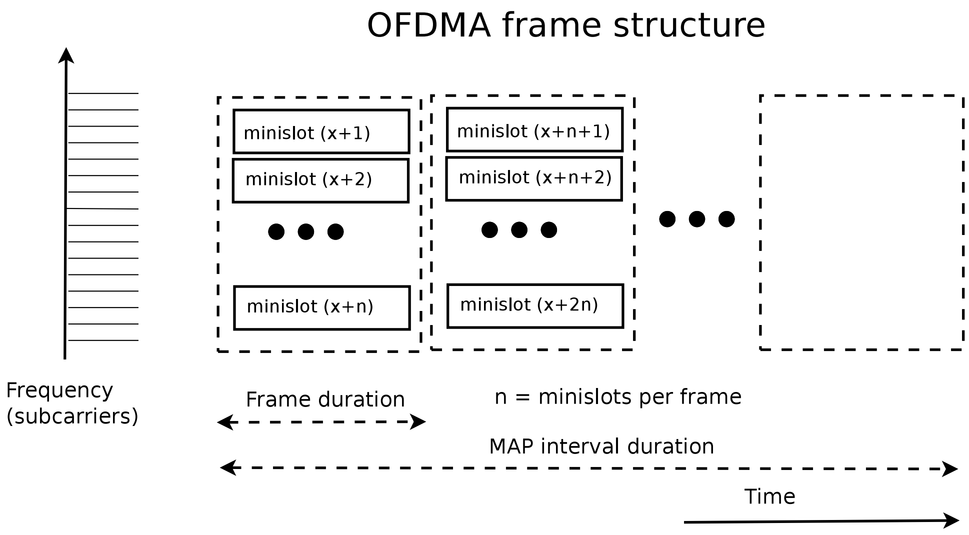

Fig. 1: DocsisNetDevice OFDMA model

The CMTS schedules grants for the CM(s) across a MAP Interval. The duration of MAP Intervals is required to be an integer number of minislots, but in this simulation model it is additionally constrained to be an integer number of OFDMA frames. Within the MAP interval, grants to CMs are composed of an integer number of minislots. Grants can span frame boundaries, but cannot span MAP Interval boundaries.

CmNetDevice pipeline illustrates that the model maintains an internal pipeline to track which data frames are eligible for transmission in which MAP interval. There is a delay between requests being calculated by the CM and the requests being considered in the formation of a MAP. When it builds a MAP, the CMTS will take into consideration all requests calculated up until the deadline. Since requests are piggybacked on grants, the timing of the grant within the MAP interval will affect whether the request beats the deadline or not. In the figure, the grant for MAP (z-2) occurs before the deadline for processing for MAP interval z, so any requests notionally sent here can be handled in MAP interval z. If the grant had happened later than the deadline, however, the request would have been scheduled in MAP interval (z+1).

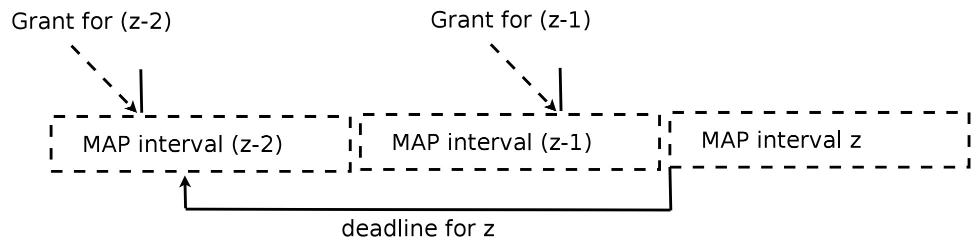

Fig. 2: CmNetDevice pipeline

The upstream transmission latency has four components, the CM_US_pipeline_factor, transmission time, propagation delay, and CMTS_US_pipeline_factor. *CmNetDevice timeline* shows the expected case that both pipeline_factors equal 1 (i.e. one frame time at each end). This pipeline factor is intended to model the burst generation processing at the transmitter and the demodulation and FEC decoding at the receiver. In the figure, the grant is shown as shaded minislots and spans two frames. Note that, if transmission latency is calculated from the start of grant, then it only includes transmission time + propagation delay + CMTS_US_pipeline_factor.

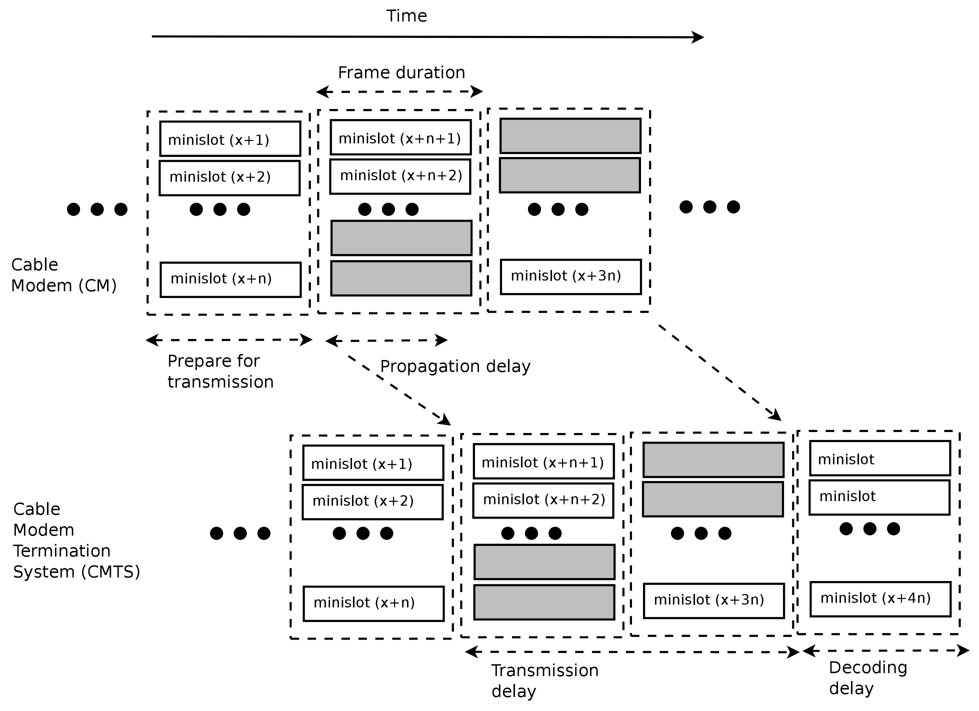

Fig. 3: CmNetDevice timeline

2.1.4 CmtsUpstreamScheduler Attributes

The key attributes governing performance are the Rate, PeakRate, and MaxTrafficBurst (controlling rate shaping) for the corresponding service flow and FreeCapacityMean and FreeCapacityVariation (controlling the amount of notional congestion). Two classes, a `ServiceFlow` and an `AggregateServiceFlow`, are used in conjunction with a dual token bucket rate shaper to regulate the granting of transmission opportunities. A cable modem may have one stand-alone service flow that has a deep queue and DOCSIS-PIE AQM, or it may have two service flows under an aggregate service flow, corresponding to a low-latency and classic service flow combination as defined by low latency DOCSIS.

The dual token bucket parameters:

- Rate (bits/sec) is the maximum sustained rate of the rate shaper
- MaxTrafficBurst (bytes) is the token bucket depth for the max sustained rate shaper
- PeakRate (bits/sec) is the peak rate of the rate shaper

These are associated with either the `ServiceFlow` (in the case of a stand-alone service flow) or with the `AggregateServiceFlow` (in the case of the low latency DOCSIS configuration). This is discussed in more detail in a later Chapter of this document.

The scheduler also supports an abstract congestion model that may constrain the delivery of grants to the cable modem.

- FreeCapacityMean (bits/sec) is the notional amount of capacity available for the CM. When multiplied by the MAP interval and divided by 8 (bits/byte) it yields roughly the amount of bytes in a MAP interval available for grant (on average).
- FreeCapacityVariation is the percentage variation on a MAP-by-MAP basis around the FreeCapacityMean to account for notional congestion.

For each MAP interval, the scheduler does a uniform random draw to calculate the amount of free capacity (minislots) in the current interval. The grant is then limited to this number of minislots.

2.2 Low Latency DOCSIS Features

When an `AggregateServiceFlow` object is instantiated (along with its two constituent `ServiceFlow` objects), this forms a “Low Latency ASF” which supports dual queue coupled AQM and queue protection.

2.2.1 Dual Queue Coupled AQM

The dual queue coupled AQM model is embedded into the CmNetDevice and CmtsNetDevice objects. Based on the IP ECN codepoint or other classifiers, packets entering the DocsisNetDevice are classified as either Low Latency or Classic, and enqueued in the respective queue. The Dual Queue implements a relationship between the ECN marking probability for Low Latency traffic and the drop probability for Classic traffic. Packets are dequeued from the dual queue either using its internal weighted deficit round robin (WDRR) scheduler that balances between the two internal queues, or based on grants that explicitly provide transmission opportunities for each of the two service classes. The implementation of Dual Queue Coupled AQM closely follows the pseudocode found in Annexes M, N, and O of [DOCSIS3.1.I19].

2.2.2 Queue Protection

QueueProtection is an additional object that inspects packets arriving to the Low Latency queue and, if the queue becomes overloaded and the latency is impacted beyond a threshold, will sanction (redirect) packets from selected flows into the Classic queue. Flows are hashed into one of 32 buckets (plus an overload bucket), and in times of congestion, flows will maintain state in the bucket and generate congestion scores that, when crossing a threshold score, will result in the packet being redirected. In this manner, the queue can be protected from certain overload situations (as might arise from misclassification of traffic), and the system tends to sanction the most heavy users of the queue before lighter users of the queue. Queue Protection is optional and can be disabled from a simulation.

The implementation of Queue Protection closely follows the pseudocode found in Annex P of [DOCSIS3.1.I19].

CHAPTER

THREE

DOCSIS SYSTEM CONFIGURATION

3.1 Upstream System & Model Parameters

3.1.1 DOCSIS Upstream OFMDA Channel Parameters

- UsScSpacing: Upstream Subcarrier Spacing. Valid values 25e3 & 50e3 (25kHz & 50kHz); default = 50e3
- NumUsSc: Number of active upstream subcarriers. Valid values: 1-1900 for 50kHz SCs, 1-3800 for 25kHz SCs; default = 1880
- SymbolsPerFrame: Number of OFDMA symbols per OFDMA frame. Valid values 6-36; default = 6
- UsSpectralEfficiency: Upstream spectral efficiency in bps/Hz. Modulation order can vary on a per-subcarrier basis, this model implements this by applying the average value to all subcarriers. Valid values 1.0 - 12.0; default = 10.0
- UsCpLen: Upstream cyclic prefix length. Valid values: 96, 128, 160, 192, 224, 256, 288, 320, 384, 512, 640; default = 256

3.1.2 DOCSIS Upstream MAC Layer Parameters

- UsMacHdrSize: Upstream MAC Header Size (bytes). Valid values: 6-246; default = 10
- UsSegHdrSize: Upstream Segment Header Size (bytes). Always 8 bytes; default = 8
- MapInterval: Target MAP Interval (seconds). The system will round this to the nearest integer multiple of the OFDMA frame duration to set the actual MAP interval used for simulation. Valid values: any ; default = 2e-3

3.1.3 Upstream Model Parameters

- CmtsMapProcTime: CMTS MAP Processing Time. The amount of time the model includes for performing scheduling operations. This includes any “guard time” the CMTS wants to factor in. Valid values: 400us is the maximum allowed per DOCSIS spec; default = 200us
- CmUsPipelineFactor: = 1; CM burst preparation time, expressed as an integer of OFDMA frame times in advance of burst transmission that CM begins encoding. Valid values: integer; default = 1
- CmtsUsPipelineFactor: = 1; CMTS burst processing time, expresssed as an integer of OFDMA frame times after burst reception completes that CMTS ends decoding. Valid values: integer; default = 1

3.2 Downstream System & Model Parameters

3.2.1 DOCSIS Downstream OFDM Channel Parameters

- DsScSpacing: Downstream Subcarrier Spacing. Valid values 25e3 & 50e3 (25kHz & 50kHz) default = 50e3
- NumDsSc: Number of active downstream subcarriers. Valid values: 1-3745 for 50e3 SC_spacing, 1-7537 for 25e3 SC_spacing. default = 3745
- DsSpectralEfficiency: Downsteam Spectral Efficiency. The downstream modulation profile which the CM is using (bps/Hz). Modulation order can vary on a per-subcarrier basis, this model implements this by applying the average value to all subcarriers. Valid range (float) 4.0–14.0; default = 12.0
- DsIntlvM: Downstream Interleaver "M". Valid values 1-32 for 50e3 SC spacing, 1-16 for 25e3 SC spacing, M=1 means "off"; default = 3
- DsCpLen: DS cyclic prefix length (Ncp). Valid values: 192,256,512,768,1024; default = 512
- NcpModulation: Next Codeword Pointer modulation order. Valid values: 2,4,6; default = 4

3.2.2 DOCSIS Downstream MAC Layer Parameters

- DsMacHdrSize: Downstream MAC Header Size (bytes). Typically 10(no channel bonding) or 16(channel bonding). Valid values: 6-246. ; default = 10

3.2.3 Downstream Model Parameters

- CmtsDsPipelineFactor: CMTS transmission processing budget. Expressed in symbol times in advance of tx that encoding begins. Valid values: integers; default = 1
- CmDsPipelineFactor: CM reception processing budget. Expressed in symbol times after rx completes that decoding completes. Valid values: integers; default = 1
- AverageCodewordFill: Factor to account for 0xFF padding bytes between frames, shortened codewords due to profile changes, etc. Valid values: 0.0-1.0; default = 0.99

3.3 System Configuration Parameters

- NumUsChannels: Number of US channels managed by this DS channel. This is used to calculate the UCD and MAP messaging overhead on the downstream channel. Valid values: integers; default = 1
- AverageUsBurst: Average size of an upstream burst (bytes), used to calculate MAP messaging overhead on the downstream channel. Valid values: integers; default = 150
- AverageUsUtilization: Average utilization of the upstream channel(s). Used to calculate MAP overhead on the downstream channel. Valid values: 0.0-1.0; default = 0.1
- MaximumDistance: Plant kilometers from furthest CM to CMTS. Max per DOCSIS spec is 80km. Valid values: 1.0-2000.0; default = 8

3.4 Implementation

These attributes are defined in `docsis-net-device.cc`.

3.5 Usage

An example usage is provided in the program `residential-example.cc` around line 1210:

```
// Configure DOCSIS Channel & System parameters

 // Upstream channel parameters
 docsis.GetUpstream (linkDocsis)->SetAttribute ("UsScSpacing", DoubleValue (50e3));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("NumUsSc", UintegerValue (1880));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("SymbolsPerFrame", UintegerValue (6));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("UsSpectralEfficiency", DoubleValue␣
↪(10.0));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("UsCpLen", UintegerValue (256));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("UsMacHdrSize", UintegerValue (10));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("UsSegHdrSize", UintegerValue (8));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("MapInterval", TimeValue␣
↪(MilliSeconds (1)));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("CmtsMapProcTime", TimeValue␣
↪(MicroSeconds (200)));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("CmtsUsPipelineFactor", UintegerValue␣
↪(1));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("CmUsPipelineFactor", UintegerValue␣
↪(1));

 // Upstream parameters that affect downstream UCD and MAP message overhead
 docsis.GetUpstream (linkDocsis)->SetAttribute ("NumUsChannels", UintegerValue (1));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("AverageUsBurst", UintegerValue␣
↪(150));
 docsis.GetUpstream (linkDocsis)->SetAttribute ("AverageUsUtilization", DoubleValue␣
↪(0.1));

 // Downstream channel parameters
 docsis.GetDownstream (linkDocsis)->SetAttribute ("DsScSpacing", DoubleValue (50e3));
 docsis.GetDownstream (linkDocsis)->SetAttribute ("NumDsSc", UintegerValue (3745));
 docsis.GetDownstream (linkDocsis)->SetAttribute ("DsSpectralEfficiency", DoubleValue␣
↪(12.0));
 docsis.GetDownstream (linkDocsis)->SetAttribute ("DsIntlvM", UintegerValue (3));
 docsis.GetDownstream (linkDocsis)->SetAttribute ("DsCpLen", UintegerValue (512));
 docsis.GetDownstream (linkDocsis)->SetAttribute ("CmtsDsPipelineFactor",␣
↪UintegerValue (1));
 docsis.GetDownstream (linkDocsis)->SetAttribute ("CmDsPipelineFactor", UintegerValue␣
↪(1));
 docsis.GetDownstream (linkDocsis)->SetAttribute ("DsMacHdrSize", UintegerValue (10));
 docsis.GetDownstream (linkDocsis)->SetAttribute ("AverageCodewordFill", DoubleValue␣
↪(0.99));
 docsis.GetDownstream (linkDocsis)->SetAttribute ("NcpModulation", UintegerValue (4));

 // Plant distance (km)
 docsis.GetUpstream (linkDocsis)->SetAttribute ("MaximumDistance", DoubleValue (8.0));
```

CHAPTER

FOUR

SERVICE FLOW CONFIGURATION

4.1 Service Flow Model Overview

Service Flows are defined in DOCSIS as a unidirectional flow of packets to which certain Quality-of-Service properties are attributed or enforced. Each Service Flow (SF) has a unique identifier called a Service Flow Identifier (SFID) assigned by the CMTS. In practice, upstream service flows also have a unique Service Identifier (SID), but the ns-3 model does not use SIDs, and instead uses the SFIDs in place of SIDs.

In DOCSIS 3.1, each cable modem would have at least one upstream and one downstream service flow defined, with rate shaping enabled, and all user traffic would typically use these two service flow definitions. Low Latency DOCSIS uses an extension of this concept to define Aggregate Service Flows which consist of paired (unidirectional) Service Flows, one for classic traffic, and one for low latency traffic. An Aggregate Service Flow (ASF) may have a single Service Flow defined or (more commonly) would have two Service Flows. The ASF has aggregate rate shaping parameters specified, and the individual SF may also have rate shaping parameters on a per-SF basis.

In the ns-3 model, only one CM is possible on a link, so the user will need to configure either an ASF or a single SF for the upstream direction, and one for the downstream direction, and will need to add these objects to the appropriate device (the CmNetDevice for the upstream ASF or SF, and the CmtsNetDevice for the downstream ASF or SF).

In a real DOCSIS system, SFIDs are unique across all of the upstream and downstream SFs within a “MAC Domain”. Currently the ns-3 model uses a fixed SFID of 1 for the classic SF or for a single standalone SF, and uses a fixed SFID of 2 for the low latency SF. These same SFIDs are used in both directions.

Configuration of SFs or ASFs is essential for conducting a DOCSIS simulation and should be performed after the DOCSIS topology has been constructed.

4.2 Implementation

The implementation can be found in `docsis-configuration.h`. Two classes are defined: `ServiceFlow` and `AggregateServiceFlow`. The members of these objects are organized according to Annex C of [DOCSIS3.1.I19]; in practice, the elements of SF and ASF configuration are encoded in TLV structures, and the ns-3 representation of them closely mirrors the TLV structure and naming convention found in the specification.

The classes `ServiceFlow` and `AggregateServiceFlow` derive from the ns-3 base class `Object`, which means that they are heap-allocated objects managed by the ns-3 smart pointer class `Ptr`.

The class definition is more like a C-style struct with public data members than a C++ class with private data member accessed by methods. In addition, an inheritance hierarchy is not defined. Instead, a number of members are defined for each class, and users selectively choose to enable the parameters of interest in their simulation. By default, the default values of each of these members will not cause any configuration actions; it is only when the user changes a value that such configuration will have an effect in the program. Parameters that do not apply to a particular service flow configuration are ignored by the model, and some are marked as 'for future use' and are not functional at this time.

4.3 Usage

Typical usage can be found in the program `residential-example.cc` around line 1170:

4.3.1 Single SF configuration

```
DocsisHelper docsis;
...
// Add service flow definitions
if (numServiceFlows == 1)
  {
    NS_LOG_DEBUG ("Adding single upstream (classic) service flow");
    Ptr<ServiceFlow> sf = CreateObject<ServiceFlow> (CLASSIC_SFID);
    sf->m_maxSustainedRate = upstreamMsr;
    sf->m_peakRate = upstreamPeakRate;
    sf->m_maxTrafficBurst = static_cast<uint32_t> (upstreamMsr.GetBitRate () *␣
↪maximumBurstTime.GetSeconds () / 8);
    docsis.GetUpstream (linkDocsis)->SetUpstreamSf (sf);
    NS_LOG_DEBUG ("Adding single downstream (classic) service flow");
    sf = CreateObject<ServiceFlow> (CLASSIC_SFID);
    sf->m_maxSustainedRate = downstreamMsr;
    sf->m_peakRate = downstreamPeakRate;
    sf->m_maxTrafficBurst = static_cast<uint32_t> (downstreamMsr.GetBitRate () *␣
↪maximumBurstTime.GetSeconds () / 8);
    docsis.GetDownstream (linkDocsis)->SetDownstreamSf (sf);
  }
```

The first block of code shows the simpler case of a single service flow in each direction. This corresponds to a DOCSIS 3.1 configuration without Low Latency DOCSIS support. The upstream SF object is first created and assigned to a smart pointer `sf`:

```
Ptr<ServiceFlow> sf = CreateObject<ServiceFlow> (CLASSIC_SFID);
```

Note here that an argument to the constructor, using a defined constant, is required. In the ns-3 model, a classic SF is assigned with a SFID of value 1 (CLASSIC_SFID), and a low latency SF is assigned with a SFID value of 2 (LOW_LATENCY_SFID). The ns-3 code uses these special values to distinguish between the two types.

Next, the rate shaping parameters are assigned. This step is always required in a simulation (to set the Maximum Sustained Rate, Peak Rate, and Maximum Traffic Burst). In this program, the Maximum Traffic Burst was configured above in units of time (`maximumBurstTime`) at the MSR, so the program converts this into units of bytes.

```
sf->m_maxSustainedRate = upstreamMsr;
sf->m_peakRate = upstreamPeakRate;
sf->m_maxTrafficBurst = static_cast<uint32_t> (upstreamMsr.GetBitRate () *␣
↪maximumBurstTime.GetSeconds () / 8);
```

No other parameters are configured, so the SF object is added to the cable modem (CmNetDevice) object, as the next statement performs by using the helper object to get the upstream device:

```
docsis.GetUpstream (linkDocsis)->SetUpstreamSf (sf);
```

The process is repeated for the downstream direction, and the SF is added to the CmtsNetDevice.

4.3.2 ASF configuration

ASF configuration takes more statements, since both an `AggregateServiceFlow` object and two constituent `ServiceFlow` objects must be create for each direction. We will look at the upstream direction only.

```
else
  {
    NS_LOG_DEBUG ("Adding upstream aggregate service flow");
    Ptr<AggregateServiceFlow> asf = CreateObject<AggregateServiceFlow> ();
    asf->m_maxSustainedRate = upstreamMsr;
    asf->m_peakRate = upstreamPeakRate;
    asf->m_maxTrafficBurst = static_cast<uint32_t> (upstreamMsr.GetBitRate () *␣
↪maximumBurstTime.GetSeconds () / 8);
    Ptr<ServiceFlow> sf1 = CreateObject<ServiceFlow> (CLASSIC_SFID);
    asf->SetClassicServiceFlow (sf1);
    Ptr<ServiceFlow> sf2 = CreateObject<ServiceFlow> (LOW_LATENCY_SFID);
    sf2->m_guaranteedGrantRate = guaranteedGrantRate;
    asf->SetLowLatencyServiceFlow (sf2);
    docsis.GetUpstream (linkDocsis)->SetUpstreamAsf (asf);
```

First, the `AggregateServiceFlow` object is created; note that the ASF ID is not required in the constructor. This value defaults to zero and is not presently used in the ns-3 model. Next, the QoS parameters around rate shaping are configured at an ASF level, similar to the code for the single SF. When an ASF is present in the model, the rate shaping will be driven from the ASF-level parameters, and not from the SF-level parameters.

Following the rate shaping configuration, two individual SF objects are created and added to the ASF object. In the Low Latency SF case, an optional parameter is configured: a Guaranteed Grant Rate to enable PGS operation. Similar configuration is performed in the downstream direction (without the GGR configuration because it does not apply for downstream).

4.3.3 Options

Besides the setting of rate shaping parameters, optional overrides on some configuration parameters in the DualQueue-CoupledAqm or QueueProtection models is possible. While its possible to use the typical ns-3 configuration techniques to configure non-default values on ns-3 Attribute values in the DualQueue and QueueProtection models, which are applied when a new object is instantiated, it is recommended instead to use the ServiceFlow or AggregateService-Flow configuration mechanism. Following instantiation, the configuration of some ASF and SF parameters offers the opportunitity to override the initially instantiated configuration. This is done by selectively changing the value of the following options away from their default value.

For example, the following parameter exists in class `ServiceFlow`:

```
uint8_t m_classicAqmTarget {0}; //!< C.2.2.7.15.2 Classic AQM Latency Target (ms);␣
↪set to non-zero value to override DualQueue default
```

If the value is left at the default of zero (i.e., if the user does not set this), the act of loading an ASF or SF into the DocsisNetDevice will not change any configuration. If, however, the user elects to change this such as follows:

```
sf = CreateObject<ServiceFlow> (CLASSIC_SFID);
sf->m_classicAqmTarget = 15;
```

then this value will be used to change the value of the parameter found in the `DualQueueCoupledAqm::ClassicAqmLatencyTarget` attribute for the object that the SF is added to (either CM or CMTS). This type of configuration is somewhat redundant with other means in ns-3 of changing attribute values, but is enabled here because it is in line operationally with how some of these parameters might be configured in practice during Service Flow definition. Note also that if the code had been:

```
sf = CreateObject<ServiceFlow> (LOW_LATENCY_SFID);
sf->m_classicAqmTarget = 15;
```

the attempted configuration of ClassicAqmLatencyTarget would have been ignored because classic AQM configuration is out of scope for a Low Latency Service Flow.

The following TLV parameters (outside of the rate shaping parameters of MSR, PeakRate, and MaximumBurst that are available for all service flow types) are available for classic Service Flows:

- `m_tosOverwrite`: C.2.2.7.9: IP Type Of Service (DSCP) Overwrite
- `m_targetBuffer`: C.2.2.7.11.4: Target Buffer (bytes)
- `m_aqmDisable`: C.2.2.7.15.1: SF AQM Disable
- `m_classicAqmTarget`: C.2.2.7.15.2: Classic AQM Latency Target

The following TLV parameters are available for low latency Service Flows:

- `m_tosOverwrite`: C.2.2.7.9: IP Type Of Service (DSCP) Overwrite
- `m_targetBuffer`: C.2.2.7.11.4: Target Buffer (bytes)
- `m_aqmDisable`: C.2.2.7.15.1: SF AQM Disable
- `m_iaqmMaxThresh`: C.2.2.7.15.4: Immediate AQM Max Threshold
- `m_iaqmRangeExponent`: C.2.2.7.15.5: Immediate AQM Range Exponent of Ramp Function
- `m_guaranteedGrantRate`: C.2.2.8.13: Guaranteed Grant Rate
- `m_guaranteedGrantInterval`: C.2.2.8.14: Guaranteed Grant Interval

The following TLV parameters are available for Aggregate Service Flows:

- `m_coupled`: COUPLED behavior of Annex M
- `m_aqmCouplingFactor`: C.2.2.7.17.5: AQM Coupling Factor
- `m_schedulingWeight`: C.2.2.7.17.6: Scheduling Weight
- `m_queueProtectionEnable`: C.2.2.7.17.7: Queue Protection Enable
- `m_qpLatencyThreshold`: C.2.2.7.17.8: QPLatencyThreshold
- `m_qpQueuingScoreThreshold`: C.2.2.7.17.9: QPQueuingScoreThreshold
- `m_qpDrainRateExponent`: C.2.2.7.17.10: QPDrainRateExponent

See the file `docsis-configuration.h` for more details about the units and behavior associated with each of these parameters.

CHAPTER

FIVE

DOCSIS TESTS

Several unit and regression tests are located in the `test/` directory. A test runner program called `test.py` is provided in the top-level ns-3 directory. Running `test.py` without any arguments will run all of the tests for ns-3. Running `test.py` with the `-s` argument allows the user to limit the test to one test suite.

- `docsis-link`: Checks transmission of single packet, point-to-point mode or DOCSIS mode, and checks point-to-point mode for multiple packets. Configures the MSR to 50 Mbps and configures bursts of varying length of line-rate packets, allocated 50% to low latency and 50% to classic service flow. Checks that the classic queue is not starved when GGR is at the MSR and peak rate > MSR (i.e. tests unused grant accounting).
- `docsis-lld`: Conducts a number of LLD-specific tests on a small test network, including the callbacks reporting on the number of grant bytes used and unused, and the exact time that packets should be received based on notional grant requests, MAP arrival times, and transmission times. Also checks the arithmetic on Time to Minislot conversions.
- `dual-queue-coupled-aqm`: This test suite performs some basic unit testing on IAQM ramp thresholds.
- `queue-protection`: This test performs some basic unit testing on the traces, on the bucket selection, and on sanctioning.

BIBLIOGRAPHY

[DOCSIS3.1] CableLabs DOCSIS specifications

[DOCSIS3.1.I19] Data-Over-Cable Service Interface Specifications, DOCSIS 3.1, MAC and Upper Layer Protocols Interface Specification, CM-SP-MULPIv3.1-I19-191016, Oct. 19, 2019

[DOCSIS-PIE] A PIE-Based AQM for DOCSIS Cable Modems