The ns-3 DOCSIS® Module Documentation

Release v1.2

CableLabs

Nov 01, 2021

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 Windows 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 most are not actively maintained. Some extensions to *ns-3* can be found in the ns-3 App Store.

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 *ns3.35* release (October 2021) 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 18.04, 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 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 to disable optimizations or use *ns-3* logging and asserts, 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 example DOCSIS simulation programs. Presently, three examples are provided:

- `residential-example.cc`
- `simple-docsislink.cc`
- `docsis-configuration-example.cc`

In addition, the `experiments/` directory contains bash scripts to automate the running and plotting of scenarios. The `experiments/residential/` contains plotting and execution scripting around `residential-example.cc`. The `experiments/simple-docsislink/` contains plotting and execution scripting around `simple-docsislink.cc`.

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 the residential example program is found in the same experiments 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 and SC-QAM channels are 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 Proactive Grant Service (PGS). Supported QoS Parameters are: Maximum Sustained Traffic Rate, Peak Traffic Rate, Maximum Traffic Burst, and Guaranteed Grant Rate. The PGS scheduler supports only two values for Guaranteed Grant Interval: GGI = MAP interval & GGI = OFDMA Frame interval; Guaranteed Request Interval is not supported. The model 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 to 2ms). Before the beginning of each MAP interval, the cable modem receives a bandwidth allocation MAP message possibly providing grants for its service flows during an upcoming 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 message by calculating how many minislots that it will grant to each service flow and by scheduling the grants within the MAP interval. 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 the MAP message arrival at the CM.

2. At MAP message arrival, the CM parses the MAP message and creates a simulation event shortly before each grant time. If there is no grant for a service flow within the MAP interval, a contention request opportunity is scheduled in the next interval.
3. Upon each grant event, the CM dequeues packets as necessary from the queue, calculates any new requests (for piggybacking), notionally builds the L2 transmission frame, and schedules simulation 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 upstream scheduler.

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.

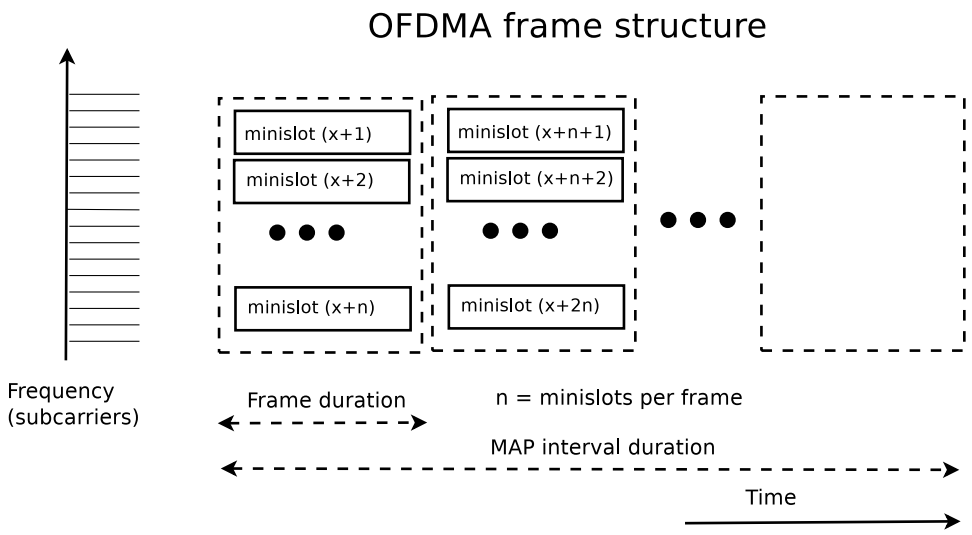

Fig. 1: DocsisNetDevice OFDMA model

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.

For any given MAP interval, a number of simulation events precede the start of the interval. Assume for generality that a given MAP interval contains one or more grants for the CM, and the list of grants will be sent to the CM in a MAP message with an Alloc Start Time (minislot number) for the start of the interval, as well as an Ack Time. Per the specification, the MAP message must be delivered to the CM at least $CM\ MAP\ processing\ time$ before the start of the MAP interval. For OFDMA, this quantity is below (per Annex B), where `K` is the number of symbols per OFDM frame:

$$600 + [(symbol\ duration + cyclic\ prefix\ duration) * (K + 1)] \text{ [usec]}$$

In the *ns-3* model, class `DocsisNetDevice` contains an attribute `CmMapProcTime` that uses the following equivalent calculation:

$$600 + [(frame\ duration) * \frac{(K + 1)}{K}] \text{ [usec]}$$

The MAP message is scheduled to arrive at `CmMapProcTime` before the start of the corresponding MAP interval. In practice, a CMTS may deliver such a message earlier than this deadline, but the *ns-3* model will always deliver it at that time.

Figure *Timeline of key events around MAP message generation* illustrates the `CmMapProcTime` in relation to the Alloc Start Time; the default value in *ns-3* is derived from other parameters and is on the order of 757 us by default. We next describe events that precede the MAP message arrival.

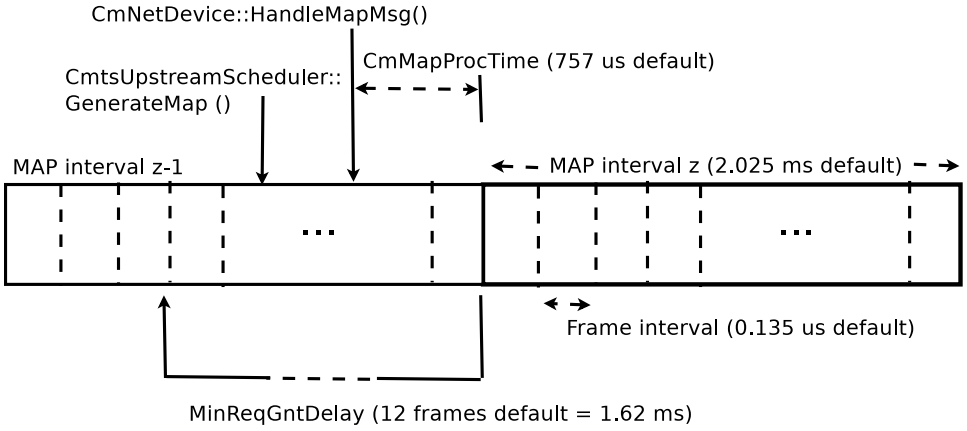

Fig. 2: Timeline of key events around MAP message generation

The MAP message is originated by the `CmtsUpstreamScheduler` at regular intervals coinciding with the notional times that would result in their on-time delivery before the Alloc Start Time of the respective MAP intervals. Conceptually, the MAP message is generated based on a regular interrupt on the CMTS, and in *ns-3*, it is generated on a regularly scheduled recurring event. The time of MAP generation, with reference to the AllocStartTime, is when the `GenerateMap()` method is called, and is calculated to be:

$$MAPgenerationtime = AllocStartTime - CmMapProcTime -$$
$$(DsIntlvDelay + 3 * DsSymbolTime + \frac{Rtt}{2} + CmtsMapProcTime) \text{ [minislots]}$$

with the quantity in parentheses represents additional processing delay (`CmtsMapProcTime`, defaulting to 200 usec) and encoding/decoding/transmission delays.

At the above MAP generation time, the `CmtsUpstreamScheduler` must determine the Ack Time and determine the grants based on received requests from the CM. In a real system, the CMTS would be decoding minislots and can determine Ack Time from that stream, but in *ns-3*, the simulation object does not observe received minislots (there are no actual minislots), and cannot reliably use the latest request time, so an estimate must be made. The simulator

calculates a quantity (in `class DocsisNetDevice`) called the `MinReqGntDelay`. This value is the deadline for any generated CM requests to notionally arrive at the CMTS for consideration in the next MAP message. The `MinReqGntDelay` is also used to populate the Ack Time in the MAP message; i.e. the Ack Time is set as:

$$AckTime = AllocStartTime - MinReqGntDelay \text{ [minislots]}$$

The `MinReqGntDelay` has units of frame intervals in *ns-3* and must be converted to minislots in the above equation.

Finally, the simulator will schedule grant requests (either piggybacked on existing grants, or sent in the request contention area) to arrive on the `CmtsUpstreamScheduler` at the corresponding time, taking into account the encoding and transmission delays. Specifically, the request delay is set as:

$$\frac{Rtt}{2} + FrameDuration * (CmUsPipelineFactor + CmtsUsPipelineFactor + 1)$$

The two pipeline factors are the frame times in advance of a burst that the CM begins encoding, and the frame times after a burst completes that the CMTS ends decoding, respectively. Both factors default to one frame interval.

CmNetDevice pipeline provides a simplified summary of the preceding figure, focusing on the following point. *ns-3* will piggyback requests onto existing grants when possible, only using a contention slot when no grant exists for a service flow. The grants to be piggybacked on may be scheduled at random times in the MAP interval. Because the `MinReqGntDelay` sets a deadline for requests to be (notionally) sent from the CM to the CMTS, if the grant (or contention request slot) happens too late in the MAP interval, it may miss a deadline and have to be scheduled in the next MAP interval. 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 $(z+1)$. The grant for $(z-1)$ occurs later in the MAP interval, so the piggybacked request will not arrive in time to be scheduled in MAP interval $(z+1)$.

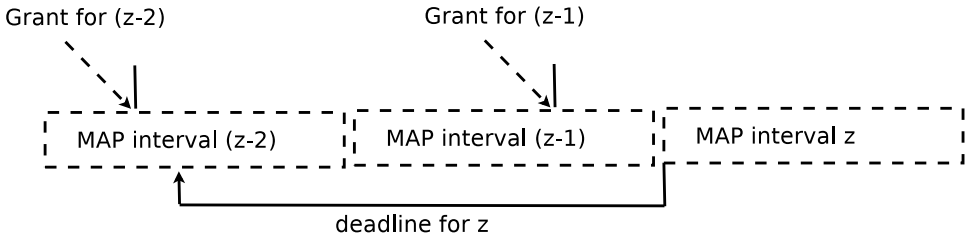

Fig. 3: CmNetDevice pipeline

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 $transmission\ time + propagation\ delay + CMTS\ US\ pipeline\ factor$.

2.1.4 CMTS Granting Behavior

In this *ns-3* model, grants for upstream traffic are created by the `CmtsUpstreamScheduler` object, on the basis of Best Effort transmission requests (as constrained by QoS parameters) and (optionally) Proactive Grant Service (PGS) parameters expressed in the Low Latency Service Flow definition.

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

- `FreeCapacityMean` (an ns-3 DataRate value) 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).

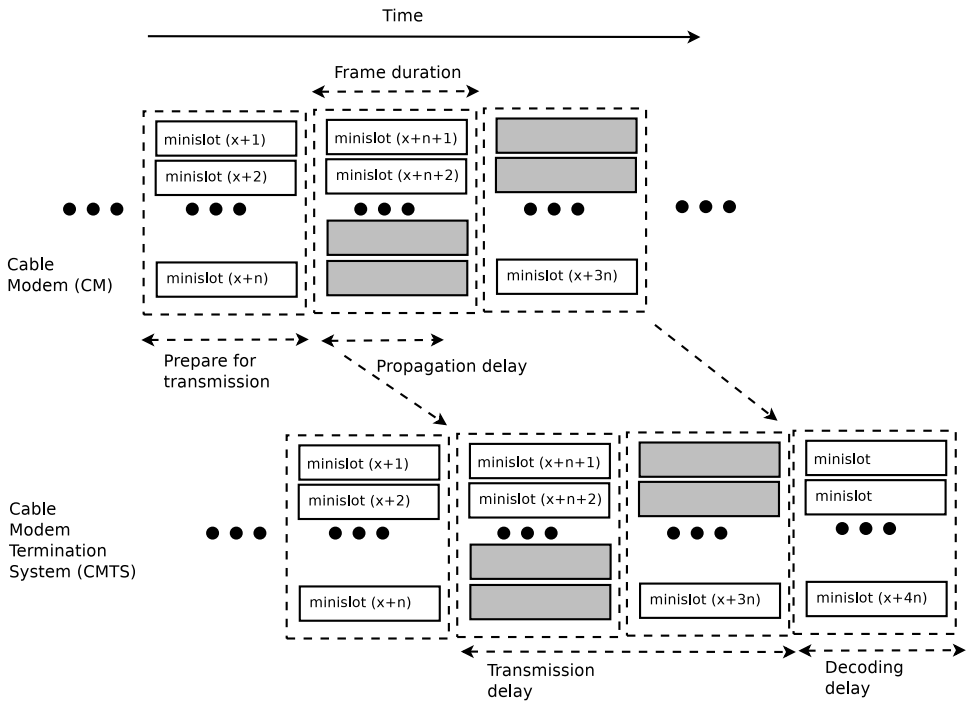

Fig. 4: CmNetDevice timeline

- `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.

The CMTS upstream scheduler regularly creates a MAP message containing data grants, and schedules the arrival of the MAP message on the `CmNetDevice` at a future simulation time based on the configured MAP message propagation and decoding time, and allowing for notional processing delays. The grants within the MAP message identify contiguous blocks of minislots within a future MAP interval, to support future upstream transmissions.

Scheduling decisions are made on the basis of four inputs:

1. Best Effort requests that are sent by the CM as either contention-based or piggybacked requests. As these requests arrive at the scheduler, they are added to a queue (backlog) of requests.
2. QoS parameters for the Aggregate Service Flow (if aggregate), or single Service Flow.
3. PGS parameters, if defined for the Low Latency Service Flow
4. The notional congestion model described above.

In addition, the scheduler model accounts for the fact that upstream bandwidth requests will include MAC header bytes that should not count against a user's shaped data rate. The CmtsUpstreamScheduler contains an attribute *MeanPacketSize* representing the mean packet size (in bytes) to assume, and will use this to internally inflate both the MSR and Peak Rate configured in the service flow according to the below factor:

$$factor = \frac{MeanPacketSize + UpstreamMACHeaderSize}{MeanPacketSize}$$

By default, this factor is $\frac{200+10}{200} = 1.05$, meaning that if the CMTS scheduler polices the granted rates as adjusted by this factor, it will be enough to compensate for the presence fo MAC header bytes included within the data grant requests (also, rounding up to minislot byte boundaries provides a small amount of additional overage). This inflation is not just a simulation artifact; something similar must be done in actual CMTS implementations.

Another key attribute used to apportion the available grant between Low Latency and Classic Service Flows is the *SchedulingWeight* attribute, that corresponds to the parameter defined in section C.2.2.9.17.6 of the specification. More information about Inter-SF scheduling with a Weighted Round Robin (WRR) scheduler is provided in section 7.7.3.2 of the specification.

The scheduling steps taken, each MAP generation time, are as follows:

1. The number of free minislots available to possibly schedule is calculated. If the congestion model is not being used, this quantity will equal the number of minislots in the MAP interval; otherwise, it will be some lesser number of minislots.
2. If PGS is configured, PGS grants are allocated according to the specified Guaranteed Grant Rate (GGR) and Guaranteed Grant Interval (GGI). In addition, the first PGS grant will always be scheduled in the first MAP messsage that is generated following simulation start. The frame offset for this first PGS grant is an attribute *FirstPgsGrantFrameOffset* of *CmtsUpstreamScheduler*, defaulting to starting in the first scheduled frame of the first MAP message generated.

   If a GGR is set but GGI is absent, the standard says that behavior is vendor-specific. The *ns-3* code sets the GGI equal to one MAP interval.

   The CMTS upstream scheduler will schedule minislots such that the equation for the number of bytes granted, defined in section C.2.2.10.13 of the specification, is followed. Because segment headers are not modelled in *ns-3*, the equation simplifies to

$$The code does not enforce that PGS grants must fit within the free minislots when the congestion model is being used (i.e., the conge$$

3. The number of rate shaped bytes available (according to the token bucket shaper) is calculated. The ASF-level QoS parameters (or if a single Classic Service Flow, the SF-level QoS parameters) are used, as inflated by the *MeanPacketSize* attribute configuration defined above. The special value of zero MSR will lead to no shaping of grant requests.
4. The number of additional (i.e., in addition to PGS grants) minislots to be granted (possibly across two service flows) is calculated as follows. Recall that the free minislots and available bytes due to token bucket shaping may constrain the overall grant. Within this constraint, the scheduling weight is used to provide a weighted allocation of rate-shaped bytes. If one service flow is not able to fill its weighted portion of the rate shaped bytes, and the other service flow requests bytes that exceed its weighted portion, the other service flow is permitted to send more bytes so long as the total available minislots and rate-shaped bytes are not exceeded. Each service flow's allocated best-effort bytes are converted to minislots (the minimum number of minislots needed to handle the bytes, by rounding up).

The above calculations will yield the number of minislots granted to one or both service flows for the MAP interval being scheduled, to satisfy, as much as possible, the requests that the scheduler has received. If PGS is configured, then the PGS minislots (already allocated) will set the floor on the number of minislots to be granted; best-effort minislots will be added to these PGS minislots as needed to satisfy the requests, such that the scheduling weight and available minislots are respected. The next step is to schedule the distribution of data grants within the MAP interval, depending on the scheduling weight and taking into account any PGS grants.

The actual grant assignments are made as follows. If PGS grants exist in the MAP interval, any additional best-effort minislots that must be allocated to the low latency service flow are tacked onto the existing PGS grant or grants, and the classic grant (if any) is then scheduled by searching for the first set of minislots (from the front of the MAP interval) that can accommodate the grant (creating multiple grants if necessary).

If PGS grants do not exist, either one or two best-effort grants will be scheduled as one contiguous block of minislots, at a starting minislot that ensures that the grant ends within the same MAP inteval. In this case, the `SchedulerRandomVariable` is used to pick the randomized starting minislot number. By default, this is a uniform random variable, but can be changed to another distribution or set to an ns-3 "constant" random variable to make the assignment deterministic.

2.1.5 CmtsUpstreamScheduler Granting Implementation

The scheduling process for the `CmtsUpstreamScheduler`, described above, is implemented as follows. This object is responsible for periodically (once per MAP interval) generating a MAP message towards the CM with scheduled grants. The timing of this event is dictated by various other constants in the model (described above in the CmNetDevice timeline figure); this section describes how the above granting behavior is implemented.

The periodic method that is called is `CmtsUpstreamScheduler::GenerateMap()`. We next review this method line-by-line.

```
283 void
284 CmtsUpstreamScheduler::GenerateMap (void)
285 {
286   NS_LOG_FUNCTION (this);
287
288   InitializeMapReport ();
289   // Calculate number of minislots available.  The congestion model (if
290   // active) is used in CalculateFreeCapacity ().  The number of free
291   // minislots may be used in the PGS scheduling task (see code below
292   // that is disabled by default) and is used in the best effort scheduling
293   // task.
294   uint32_t freeMinislots = CalculateFreeCapacity ();
295   NS_LOG_DEBUG ("Free minislots: " << freeMinislots << "; total: " << GetUp
↪stream ()->GetMinislotsPerMap ());
```

The `InitializeMapReport()` (line 288) records some initial state in the MAP report trace that is generated at the end of this method. Next, free capacity is calculated (line 294). If there is no congestion model, the number of free minislots returned will be equal to the total number of minislots per MAP. The variable *freeMinislots* is an upper bound on the maximum number of minislots that can be granted.

If configured, PGS grants are next handled. The granting behavior dictated by the GGR and GGI will result in the first grants made in the MAP schedule (if necessary to conform to the GGI), and after any PGS grant scheduling, any best effort grants will be scheduled. A number of variables are configured upon scheduler start (*CmtsUpstreamScheduler::Start()*), as follows. The variables are declared in the header file:

```
uint32_t m_pgsFrameInterval {0}; // Frame interval corresponding to GGI
uint32_t m_pgsMinislotsPerInterval {0}; // PGS minislots to grant each GGI
uint32_t m_excessPgsBytesPerInterval {0}; // Excess PGS bytes granted each GGI
```

The first variable, *m_pgsFrameInterval*, is the frame interval that corresponds to the GGI, which is not expressed in units of frames but in units of time. For example, the frame interval may be 235 microseconds. If the GGI is specified (in the service flow definition) as 235 microseconds, then the *m_pgsFrameInterval* will be set to 1. If the GGI is set lower than this frame interval, the simulation will exit with an error. Otherwise, if the GGI is set to a number of microseconds that is not an integer multiple of the frame interval, it will be rounded down. Continuing with the example, if the GGI is specified in the range 236-269 microseconds, *m_pgsFrameInterval* will also be set to 1. If in the range 270-404 microseconds, *m_pgsFrameInterval* will be set to 2, and so on.

Given the frame interval needed to meet the GGI, the next variable that is set is the number of bytes per interval needed to satisfy the bytes granted according to section C.2.2.10.13 of the specification. In general, this will not equal a round number of minislots, so the number of minislots needed per grant interval is calculated (*m_pgsMinislotsPerInterval*). Finally, any excess bytes that resulted from the roundup to next largest number of minislots is stored in (*m_excessPgsBytesPerInterval*). This excess is not used to influence grant assignment but is reported in the MAP message trace.

The above three variables are used in the code from *GenerateMap()* that follows.

```
300   uint32_t pgsGrantedBytes = 0;
301   uint32_t excessPgsGrantedBytes = 0;  // Track overage due to minislot roundup
302                                        // excessPgsGrantedBytes is reported
303                                        // in the MAP report but not used herein
304   // Maintain grants in a notional list, but one that is implemented as a
305   // C++ map, with the key to this map being the grant start time (in minislots)
306   // Key to this map is the grant start time (in minislots)
307   std::map<uint32_t, Grant> pgsGrantList;
308   uint32_t pgsMinislots = 0;
309   if (m_llSf && m_llSf->m_guaranteedGrantRate.GetBitRate () > 0)
310     {
311       // m_allocStartTime is the first minislot in this MAP interval being
312       // scheduled.  nextAllocStartTime is the first minislot in the next
313       // MAP interval being scheduled (in the next scheduling cycle)
314       SequenceNumber32 nextAllocStartTime = m_allocStartTime
315         + GetUpstream ()->TimeToMinislots (GetUpstream ()->GetActualMapInterval␣
↪());
316       NS_LOG_DEBUG ("AllocStartTime: " << m_allocStartTime << ";␣
↪nextAllocStartTime: " << nextAllocStartTime);
317       uint32_t pgsMinislotsAllowed = freeMinislots;
318       // Determine each frame that requires a PGS grant (if any), and add grant
319       // Maintain time quantities in units of minislots
320       for (SequenceNumber32 i = m_lastPgsGrantTime + m_pgsFrameInterval *␣
↪GetUpstream ()->GetMinislotsPerFrame ();
321         i < nextAllocStartTime && pgsMinislotsAllowed; )
322         {
323           NS_LOG_DEBUG ("PGS grant of " << m_pgsMinislotsPerInterval << " at␣
↪minislot offset "
324             << (i - m_allocStartTime));
325           m_lastPgsGrantTime = i;
326           // Add the grant
327           Grant lGrant;
328           lGrant.m_sfid = LOW_LATENCY_SFID;
329           if (m_pgsMinislotsPerInterval < pgsMinislotsAllowed)
330             {
331               lGrant.m_length = m_pgsMinislotsPerInterval;
332               pgsMinislots += m_pgsMinislotsPerInterval;
333 #if 0
334               // See below
335               pgsMinislotsAllowed -= m_pgsMinislotsPerInterval;
336 #endif
337               pgsGrantedBytes += m_pgsBytesPerInterval;
338               excessPgsGrantedBytes += m_excessPgsBytesPerInterval;
339             }
340 #if 0
341           // This code branch exists in case the model is changed to
342           // constrain PGS grants by congestion state.  Presently, the
343           // model will not constrain PGS grants by congestion state.
344           else
345             {
346               NS_LOG_DEBUG ("PGS grants limited by free minislots");
347               lGrant.m_length = pgsMinislotsAllowed;
348               pgsMinislots += pgsMinislotsAllowed;
349               pgsGrantedBytes += pgsMinislotsAllowed * GetUpstream ()->
↪GetMinislotCapacity ();
350               pgsMinislotsAllowed = 0;
351             }
```

(continues on next page)

(continued from previous page)

```
352 #endif
353           int32_t offset = i - m_allocStartTime;
354           NS_ASSERT_MSG (offset >= 0, "Offset calculation is negative");
355           lGrant.m_offset = static_cast<uint32_t> (offset);
356           pgsGrantList.insert (std::pair<uint32_t, Grant> (i.GetValue (),␣
↪lGrant));
357           i += (m_pgsFrameInterval * GetUpstream ()->GetMinislotsPerFrame ());
358         }
359       NS_LOG_DEBUG ("PGS grants: " << pgsGrantList.size () << "; bytes: " <<␣
↪pgsGrantedBytes
360         << "; excess bytes granted: " << excessPgsGrantedBytes);
```

Each time a PGS grant is made, the variable *m_lastPgsGrantTime* stores the minislot number of the first minislot in the frame. This state is saved across MAP intervals, so that if the GGI is greater than one MAP interval, the correct first frame (if any) for a PGS grant in this MAP interval can be used. The loop from lines 320-358 will allocate PGS grants as needed between AllocStartTime and the next AllocStartTime. The grants are temporarily inserted into a list called *pgsGrantList*. Two counters are maintained for later use: *pgsGrantedBytes* and *excessPgsGrantedBytes*.

There are also two blocks of code that are commented out; lines 333-336 and lines 340-352. This code is disabled by default but can be enabled if the user would like to also constrain the PGS grants by congestion state.

Next, the token bucket counters are incremented, and the available tokens (msr and peak tokens) are calculated and returned as a pair of values (line 369):

```
367   // Update token bucket rate shaper
368   // tokens->first == MSR, tokens->second == peak
369   std::pair<int32_t, uint32_t> tokens = IncrementTokenBucketCounters ();
370   uint32_t availableTokens = 0;
371   if (tokens.first <= 0)
372     {
373       // This implementation will not grant if there is a deficit
374       NS_LOG_DEBUG ("MSR tokens less than or equal to zero; no grant will be made
↪");
375     }
376   else
377     {
378       availableTokens = std::min (static_cast<uint32_t> (tokens.first), tokens.
↪second);
379       NS_LOG_DEBUG ("Available tokens: " << availableTokens);
380     }
```

The method *IncrementTokenBucketCounters ()* replenishes the MSR token bucket based on the configured AMSR (or MSR if a single service flow), and sets the peak token bucket based on the peak rate (number of bytes that can be sent in one MAP interval at the peak rate). Line 378 sets the available tokens to the minimum of the MSR and peak tokens. This is a further cap on the number of bytes that can be granted.

The next block of code works on best effort requests, if any.

```
385   uint32_t cRequestBacklog = m_requests[m_classicSf->m_sfid];
386   uint32_t lRequestBacklog = 0;
387   if (m_llSf)
388     {
389       lRequestBacklog = m_requests[m_llSf->m_sfid];
390     }
391   NS_LOG_DEBUG ("cRequestBacklog: " << cRequestBacklog << "; lRequestBacklog: " <
↪< lRequestBacklog);
```

The number of unserved requests have been stored in the *m_requests* array, and are copied to local variables here (units are bytes).

```
396   // Perform grant allocation for remaining best-effort backlog
397   uint32_t cAllocatedMinislots = 0;
398   uint32_t lAllocatedMinislots = 0;
399   if (availableTokens > 0 && (cRequestBacklog || lRequestBacklog))
400     {
401       // Bounded by freeMinislots and availableTokens, determine how much of the
402       // classic and low latency backlog can be served in this MAP interval.  If␣
↪PGS
403       // is being used, account for the bytes and minislots already allocated.
404       // This method is where the scheduling weight between the two queues
405       // is applied.
406       std::pair<uint32_t, uint32_t> allocation = Allocate (freeMinislots,␣
↪pgsMinislots, availableTokens,
407         cRequestBacklog, lRequestBacklog);
408       NS_LOG_DEBUG ("Scheduler allocated " << allocation.first << " bytes to SFID␣
↪1 and "
409         << allocation.second << " to SFID 2");
410       cAllocatedMinislots = allocation.first;
411       lAllocatedMinislots = allocation.second;
412     }
```

Line 399 checks whether the conditions exist for any best-effort grants to be allocated. First, there must be tokens available from the rate shaper, and second, there must be some best-effort request backlog. If either condition is false, no further grant allocations are performed and the grant allocation will either consist of the PGS grants (if present from above) or none at all. The *Allocate()* method (line 406) takes the existing PGS grant schedule into account and tries to allocate the remaining best-effort backlog, constrained by the limits of available free minislots and shaped bytes. The pair of values returned are the number of minislots allocated to each service flow for the MAP interval (not the exact grant schedule, but simply the number of minislots to be scheduled). The following listings go into *Allocate()* in more detail.

```
876 std::pair<uint32_t, uint32_t>
877 CmtsUpstreamScheduler::Allocate (uint32_t freeMinislots, uint32_t pgsMinisl    ␣
↪ots, uint32_t availableTokens,
878   uint32_t cRequestedBytes, uint32_t lRequestedBytes)
879 {
880   NS_LOG_FUNCTION (this << freeMinislots << pgsMinislots << availableTokens      <
↪< cRequestedBytes << lRequestedBytes);
881   NS_ASSERT_MSG ((availableTokens > 0 && (cRequestedBytes || lRequestedByte    ␣
↪s)),
882     "Allocate called with invalid arguments");
883
884   // First determine the allocation constraints.  Then try
885   // to apportion minislots to classic and LL flows according
886   // to scheduling weight.
887
888   uint32_t availableMinislots = std::min<uint32_t> (MinislotCeil␣
↪(availableTokens), freeMinislots);
889
890   uint32_t maxCMinislots = MinislotCeil (cRequestedBytes);
891   uint32_t maxLMinislots = std::max<uint32_t> (MinislotCeil (lRequestedBytes),␣
↪pgsMinislots);
892   uint32_t maxRequestedMinislots = maxCMinislots + maxLMinislots;
893   NS_LOG_DEBUG ("Available minislots (due to congestion or token bucket): " <<␣
↪availableMinislots
```

(continues on next page)

(continued from previous page)

```
894     << "; maximum requested: " << maxRequestedMinislots);
895   uint32_t minislotsToAllocate = std::min<uint32_t> (availableMinislots,␣
↪maxRequestedMinislots);
896   // At least PGS minislots must be allocated
897   minislotsToAllocate = std::max<uint32_t> (minislotsToAllocate, pgsMinislots);
```

As indicated by the comment in lines 884-886, the first block of statements calculates how many minislots would correspond to the allocation if all of the requested best-effort bytes were to be handled, to compare against the available minislots that are constrained by *freeMinislots* and the rate shaper. The number of minislots to allocate is bounded by the minimum of the minislots needed to satisfy the request backlog and the available minislots, subject to the floor of the PGS minislots that have already been allocated.

```
899   // Allocate according to the following constraints
900   // - pgsMinislots, if present, must service LL backlog
901   // - respect scheduler weight as much as possible
902   uint32_t cMinislotsAllocated = 0;
903   uint32_t lMinislotsAllocated = 0;
904   if (pgsMinislots >= maxLMinislots)
905     {
906       NS_LOG_DEBUG ("PGS minislots exceed the L request backlog");
907       if (availableMinislots > pgsMinislots)
908         {
909           cMinislotsAllocated = std::min<uint32_t> ((availableMinislots -␣
↪pgsMinislots), maxCMinislots);
910         }
911       lMinislotsAllocated = pgsMinislots;
912       NS_LOG_DEBUG ("C minislots allocated: " << cMinislotsAllocated << " L␣
↪minislots allocated: "
913         << lMinislotsAllocated);
914       return std::make_pair (cMinislotsAllocated, lMinislotsAllocated);
915     }
```

Lines 904-915 handle the case in which the PGS allocation exceeds the request backlog for the low latency service flow. Line 907 determines whether there can be any minislots allocated to the classic service flow in this case; the minimum of the classic request backlog and the remaining available minislots is used, and these quantities returned from the method.

The next case considered is when the PGS minislots already allocated are not sufficient to handle the low latency service flow backlog.

```
916   // pgsMinislots < maxLMinislots, so try to allocate more than pgsMinislots to␣
↪the L service flow
917   uint32_t lWeightedMinislotsToAllocate = static_cast<uint32_t> (static_cast
↪<double> (minislotsToAllocate)
918     * m_schedulingWeight / 256);
919   lWeightedMinislotsToAllocate = std::max<uint32_t> (lWeightedMinislotsToAllocate,
↪ pgsMinislots);
920   NS_ASSERT_MSG (minislotsToAllocate >= lWeightedMinislotsToAllocate, "Arithmetic␣
↪overflow");
921   uint32_t cWeightedMinislotsToAllocate = minislotsToAllocate -␣
↪lWeightedMinislotsToAllocate;
```

Line 917 computes the weighted number of minislots to allocate according to the scheduling weight. In case the number of low latency minislots calculated in line 917 is fewer than the number of PGS minislots, the number is pulled up (line 919). Line 921 calculates the resulting number of classic minislots to allocate, according to the weight and the PGS constraint. At this point, these two quantities (*lWeightedMinislotsToAllocate* and *cWeightedMinislotsToAllocate*) represent the allocation that will be made, so long as each service flow has a request backlog at least as large, respec-

tively. The next branches of code handle the various cases in which the request backlog does not require the weighted amount, in which case some minislots can be allocated back to the other service flow if needed (lines 922-950 below).

```
922   if (maxLMinislots <= lWeightedMinislotsToAllocate && maxCMinislots <=␣
↪cWeightedMinislotsToAllocate)
923     {
924       NS_LOG_DEBUG ("All minislot requests can be allocated");
925       cMinislotsAllocated = maxCMinislots;
926       lMinislotsAllocated = maxLMinislots;
927     }
928   else if (maxLMinislots > lWeightedMinislotsToAllocate && maxCMinislots > ␣
↪cWeightedMinislotsToAllocate)
929     {
930       NS_LOG_DEBUG ("Both minislots allocations limited; using weight of " << m_
↪schedulingWeight);
931       cMinislotsAllocated = cWeightedMinislotsToAllocate;
932       lMinislotsAllocated = lWeightedMinislotsToAllocate;
933     }
934   else if (maxLMinislots <= lWeightedMinislotsToAllocate && maxCMinislots >␣
↪cWeightedMinislotsToAllocate)
935     {
936       NS_LOG_DEBUG ("C minislots possibly limited");
937       NS_ASSERT_MSG (minislotsToAllocate > maxLMinislots, "Arithmetic overflow");
938       cMinislotsAllocated = std::min<uint32_t> ((minislotsToAllocate -␣
↪maxLMinislots), maxCMinislots);
939       lMinislotsAllocated = maxLMinislots;
940     }
941   else if (maxLMinislots > lWeightedMinislotsToAllocate && maxCMinislots <=␣
↪cWeightedMinislotsToAllocate)
942     {
943       NS_LOG_DEBUG ("L minislots possibly limited");
944       NS_ASSERT_MSG (minislotsToAllocate > maxCMinislots, "Arithmetic overflow");
945       cMinislotsAllocated = maxCMinislots;
946       lMinislotsAllocated = std::min<uint32_t> ((minislotsToAllocate -␣
↪maxCMinislots), maxLMinislots);
947     }
948   NS_LOG_DEBUG ("C minislots allocated: " << cMinislotsAllocated << " L minislots␣
↪allocated: " << lMinislotsAllocated);
949   return std::make_pair (cMinislotsAllocated, lMinislotsAllocated);
950 }
```

Returning to the `GenerateMap()` method, after *Allocate()* returns, the scheduler has already made PGS grants (if necessary) and knows the number of minislots that must be scheduled to each service flow. The next step is to build the grant list for the best effort grants, and combine them with the PGS grants if present.

```
415   // Build the grant list
416   // We use a std::map functionally as an ordered std::list.  The map key
417   // is the offset, the value is the grant.  Later, we use this map as an
418   // ordered list; we only need to iterate on it and fetch the values
419   // in order, but we exploit the property of a map that entries are stored
420   // in non-descending order of keys.
421   std::map<uint32_t, Grant> grantList;
422   if (cAllocatedMinislots || lAllocatedMinislots || pgsGrantList.size () > 0)
423     {
424       std::pair<uint32_t, uint32_t> minislots = ScheduleGrant (grantList,␣
↪pgsGrantList, pgsMinislots,
425         cAllocatedMinislots, lAllocatedMinislots);
426       uint32_t cMinislots = minislots.first;
```

(continues on next page)

(continued from previous page)

```
427       uint32_t lMinislots = minislots.second;  // includes PGS granted minislots
```

The actual grant allocation happens in `ScheduleGrant()`, which takes the (initially empty) grantList, the previously allocated pgsGrantList, and the number of classic and low latency minislots to allocate. This method (line 424) returns the modified grantList (which is passed in by reference).

ScheduleGrant() implements the following policy. If PGS grants exist, the remaining low latency minislots are tacked onto the first PGS grant. Then, the classic grant is scheduled to start at the first available minislot so as to not overlap with any low latency grant.

If there are no PGS grants in the MAP interval, both the classic and low latency grants are scheduled to start somewhere (randomly) in the frame, according to the scheduler random variable, first with the low latency minislots, followed immediately by the classic minislots, according to the scheduler random variable.

The remaining statements perform accounting based on the result of the scheduling.

```
431       // We granted possibly more bytes than requested by rounding up to
432       // a minislot boundary.  Also, PGS service may cause overallocation.
433       // Subtract the total bytes granted (including roundup) from the token
434       // bucket; unused bytes will be redeposited later based on unused grant
435       // tracking.
436       DecrementTokenBucketCounters (MinislotsToBytes (cMinislots) +␣
↪MinislotsToBytes (lMinislots));
437       // Subtract the full number of bytes granted from the request backlog
438       if (cMinislots && (m_requests[m_classicSf->m_sfid] > MinislotsToBytes␣
↪(cMinislots)))
439         {
440           NS_LOG_DEBUG ("Allocated " << MinislotsToBytes (cMinislots)
441             << " to partially cover classic request of " << m_requests[m_
↪classicSf->m_sfid]);
442           m_requests[m_classicSf->m_sfid] -= MinislotsToBytes (cMinislots);
443         }
444       else if (cMinislots && (m_requests[m_classicSf->m_sfid] <= MinislotsToBytes␣
↪(cMinislots)))
445         {
446           NS_LOG_DEBUG ("Allocated " << MinislotsToBytes (cMinislots) << " to␣
↪fully cover classic request of "
447             << m_requests[m_classicSf->m_sfid]);
448           m_requests[m_classicSf->m_sfid] = 0;
449         }
450       if (m_llSf)
451         {
452           if (lMinislots && (m_requests[m_llSf->m_sfid] > MinislotsToBytes␣
↪(lMinislots)))
453             {
454               NS_LOG_DEBUG ("Allocated " << MinislotsToBytes (lMinislots)
455                 << " to partially cover LL (non-PGS) request of " << m_requests[m_
↪llSf->m_sfid]);
456               m_requests[m_llSf->m_sfid] -= MinislotsToBytes (lMinislots);
457             }
458           else if (lMinislots && (m_requests[m_llSf->m_sfid] <= MinislotsToBytes␣
↪(lMinislots)))
459             {
460               NS_LOG_DEBUG ("Allocated " << MinislotsToBytes (lMinislots)
461                 << " to fully cover LL (non-PGS) request of " << m_requests[m_
↪llSf->m_sfid]);
462               m_requests[m_llSf->m_sfid] = 0;
```

(continues on next page)

(continued from previous page)

```
463             }
464         }
465     }
```

Line 436 subtracts the granted bytes from the token bucket counters. It is possible that minislot rounding or PGS caused overallocation, but these bytes are still deducted, because unused bytes will be redeposited later based on unused grant tracking. The request counters for both service flows are decremented if minislots have been allocated.

```
475   MapMessage mapMessage = BuildMapMessage (grantList);
476
477   // Schedule the allocation MAP message for arrival at the CM
478   Time mapMessageArrivalDelay = GetUpstream ()->GetDsIntlvDelay () + GetUpstream␣
↪()->GetDsSymbolTime ()
479     * 3 + GetUpstream ()->GetRtt () / 2 + GetUpstream ()->GetCmtsMapProcTime ();
480   NS_LOG_DEBUG ("MAP message arrival delay: " << mapMessageArrivalDelay.As␣
↪(Time::S));
481   NS_LOG_DEBUG ("Schedule MAP message arrival at " << (Simulator::Now () +␣
↪mapMessageArrivalDelay).As (Time::S));
482   m_mapArrivalEvent = Simulator::Schedule (mapMessageArrivalDelay, &
↪CmNetDevice::HandleMapMsg, GetUpstream (),
483     mapMessage);
484
485   // Trace the MAP report
486   m_mapReport.m_message = mapMessage;
487   m_mapTrace (m_mapReport);
488
489   // Reschedule for next MAP interval
490   NS_LOG_DEBUG ("Schedule next GenerateMap() at " << (Simulator::Now ()
491     + GetUpstream ()->GetActualMapInterval ()).As (Time::S));
492   m_allocStartTime += GetUpstream ()->TimeToMinislots (GetUpstream ()->
↪GetActualMapInterval ());
493   m_generateMapEvent = Simulator::Schedule (GetUpstream ()->GetActualMapInterval␣
↪(),
494     &CmtsUpstreamScheduler::GenerateMap, this);
495 }
```

Finally, the MAP message is built (line 475), scheduled to arrive at the CM (lines 482-483), and the MAP message is traced (lines 486-487). The final statements (lines 490-494) schedule the next event for `GenerateMap()`.

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.

This ns-3 model currently supports the following service flow configuration options:

1) a single Service Flow configured for a DOCSIS PIE AQM. This service flow may include rate shaping parameters (Maximum Sustained Traffic Rate, Peak Traffic Rate, and Maximum Traffic Burst) which, if explicitly configured, will cause the CM upstream requests to be rate shaped, and will cause the CMTS scheduler to rate shape the granted bytes. If rate shaping parameters are not configured, the default values will disable rate shaping.

2) a Low Latency Aggregate Service Flow with a Classic Service Flow and a Low Latency Service Flow. The Aggregate Service flow may include rate shaping QoS parameters that are enforced at the CMTS scheduler. Additionally, the Classic Service Flow may configure rate shaping parameters, in which case the CM upstream requests will be rate shaped based on Service Flow-level paramters. Although, in practice, it is possible to also configure QoS parameters on the Low Latency Service Flow, causing rate shaping of upstream grant requests, the ns-3 model does not support this; only the Classic Service Flow of a Low Latency Aggregate Service Flow may express QoS parameters.

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. 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 CMTS upstream scheduler object will enforce rate shaping based on ASF-level rate shaping parameters as part of the grant allocation process.

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). In the above example, no rate shaping parameters are configured on the Classic Service Flow, and as a result, the upstream requests will not be shaped for this service flow.

4.3.3 Options

Besides the setting of rate shaping parameters, optional overrides on some configuration parameters in the DualQueueCoupledAqm 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 AggregateServiceFlow 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