OFSwitch13 Module Documentation

Release 5.2.2

Computer Networks Laboratory at Unicamp, Brazil

Sep 01, 2023

CONTENTS

CHAPTER

ONE

MODULE DESCRIPTION

1.1 Overview

The *OFSwitch13* module enhances the ns-3 Network Simulator with Software-Defined Networking (SDN) support. Despite the fact that the *ns-3* already has a module for simulating OpenFlow switches, it provides a very outdated protocol implementation (OpenFlow 0.8.9, from 2008). Alternatively, *OFSwitch13* supports OpenFlow protocol version 1.3, bringing both a switch device and a controller application interface to the *ns-3* simulator, as depicted in *The OFSwitch13 module overview*, from [Chaves2016a]. With *OFSwitch13*, it is possible to interconnect *ns-3* nodes to send and receive traffic using `CsmaNetDevice` or `VirtualNetDevice`. The controller application interface can be extended to implement any desired control logic to orchestrate the network. The communication between the controller and the switch is realized over standard *ns-3* protocol stack, devices, and channels. The *OFSwitch13* module relies on the external BOFUSS library for OFSwitch13. This library provides the switch datapath implementation, the support for converting OpenFlow messages to/from wire format, and the `dpctl` utility tool for configuring the switch from the command line.

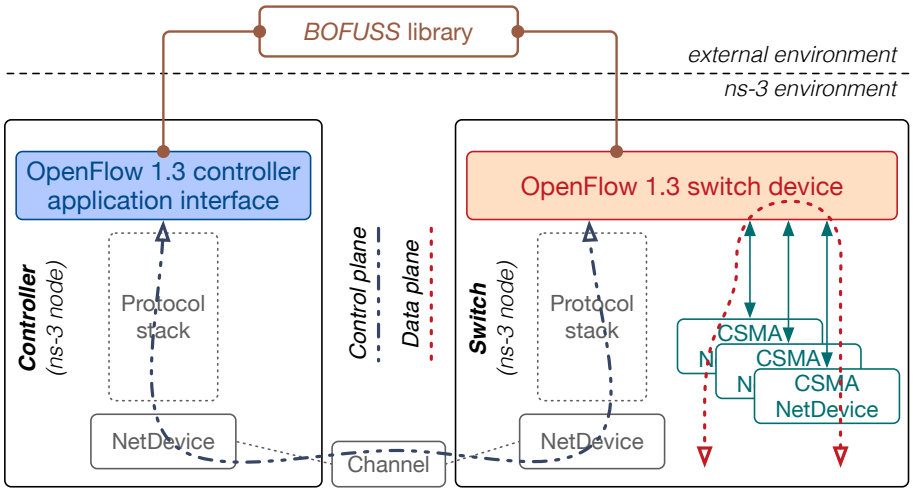

Fig. 1: The *OFSwitch13* module overview

1.2 Design

1.2.1 OpenFlow 1.3 Switch Device

The OpenFlow 1.3 switch device, namely `OFSwitch13Device`, can be used to interconnect *ns-3* nodes using the existing network devices and channels. *The OFSwitch13Device internal structure* figure shows the internal switch device structure. It takes a collection of `OFSwitch13Port` acting as input/output ports, each one associated with an *ns-3* underlying `NetDevice`. In most cases, the `CsmaNetDevice` is used to build the ports, which will act as physical ports. However, it is possible to use the `VirtualNetDevice` to implement logical ports. The switch device acts as the intermediary between the ports, receiving a packet from one port and forwarding it to another. The *BOFUSS* library provides the OpenFlow switch datapath implementation (flow tables, group table, and meter table). Thus, packets entering the switch are sent to the library for OpenFlow pipeline processing before being forwarded to the correct output port(s). OpenFlow messages received from the controller are also sent to the library for datapath configuration.

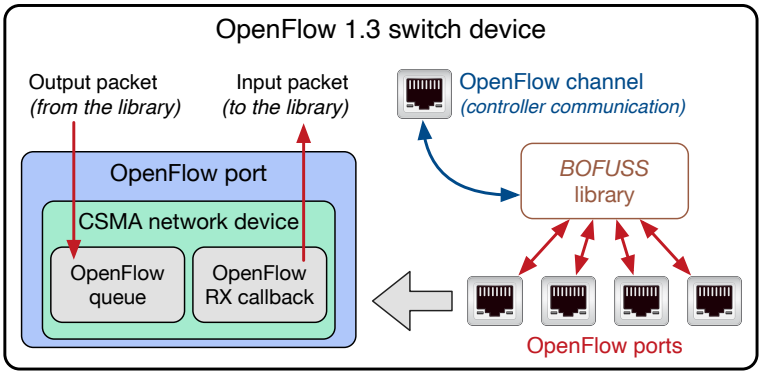

Fig. 2: The `OFSwitch13Device` internal structure

A packet enters the switch device through the OpenFlow receive callback in the underlying `NetDevice`, which is invoked for successfully received packets. This callback is a promiscuous one, but in contrast to a promiscuous protocol handler, the packet sent to this callback includes all the headers required for OpenFlow pipeline processing. Including this new callback in the `NetDevice` is the only required modification to the *ns-3* source code for *OFSwitch13* usage.

The incoming packet is checked for conformance to the CPU processing capacity (throughput) defined by the `OFSwitch13Device::CpuCapacity` attribute. Packets exceeding CPU processing capacity are dropped, while conformant packets are sent to the pipeline at the *BOFUSS* library. The module considers the concept of *virtual TCAM* (Ternary Content-Addressable Memory) to estimate the average flow table search time to model OpenFlow hardware operations. It considers that real OpenFlow implementations use sophisticated search algorithms for packet matching such as hierarchical hash tables or binary search trees. Because of that, the equation $K * log\_2\ (n)$ is used to estimate the delay, where K is the `OFSwitch13Device::TcamDelay` attribute set to the time for a single TCAM operation, and n is the current number of entries on pipeline flow tables.

Packets coming back from the library for output action are sent to the OpenFlow queue provided by the module. An OpenFlow switch provides limited QoS support employing a simple queuing mechanism, where each port can have one or more queues attached to it. Packets sent to a specific queue are treated according to that queue's configuration. Queue configuration takes place outside the OpenFlow protocol. The `OFSwitch13Queue` abstract base class implements the queue interface, extending the *ns-3* `Queue<Packet>` class to allow compatibility with the `CsmaNetDevice` used within `OFSwitch13Port` objects (`VirtualNetDevice` does not use queues). In this way, it is possible to replace the standard `CsmaNetDevice::TxQueue` attribute by this modified `OFSwitch13Queue` object. Internally, it can hold a collection of N (possibly different) queues, each one identified by a unique ID ranging from 0 to N-1. Packets sent to the OpenFlow queue for transmission by the `CsmaNetDevice` are expected to carry the `QueueTag`, which is used by the `OFSwitch13Queue::Enqueue` method to identify the internal queue that will hold the packet. Specialized `OFSwitch13Queue` subclasses can perform different output scheduling algorithms by implementing the `Peek`,

`Dequeue`, and `Remove` pure virtual methods from *ns-3* `Queue`. The last two methods must call the `NotifyDequeue` and `NotifyRemoved` methods respectively, which are used by the `OFSwitch13Queue` to keep consistent statistics.

The OpenFlow port type queue can be configured by the `OFSwitch13Port::QueueFactory` attribute at construction time. Currently, the `OFSwitch13PriorityQueue` is the only specialized OpenFlow queue available for use. It implements the priority queuing discipline for a collection of N priority queues, identified by IDs ranging from 0 to N-1 with decreasing priority (queue ID 0 has the highest priority). The output scheduling algorithm ensures that higher-priority queues are always served first. The `OFSwitch13PriorityQueue::QueueFactory` and `OFSwitch13PriorityQueue::NumQueues` attributes can be used to configure the type and the number of internal priority queues, respectively. By default, it creates a single `DropTailQueue` operating in packet mode with the maximum number of packets set to 100.

1.2.2 OpenFlow 1.3 Controller Application Interface

The OpenFlow 1.3 controller application interface, namely `OFSwitch13Controller`, provides the necessary functionalities for controller implementation. It can handle a collection of OpenFlow switches, as illustrated in *The OFSwitch13Controller internal structure* figure. For constructing OpenFlow configuration messages and sending them to the switches, the controller interface relies on the `dpctl` utility provided by the *BOFUSS* library. With a simple command-line syntax, this utility can be used to add flows to the pipeline, query for switch features and status, and change other configurations.

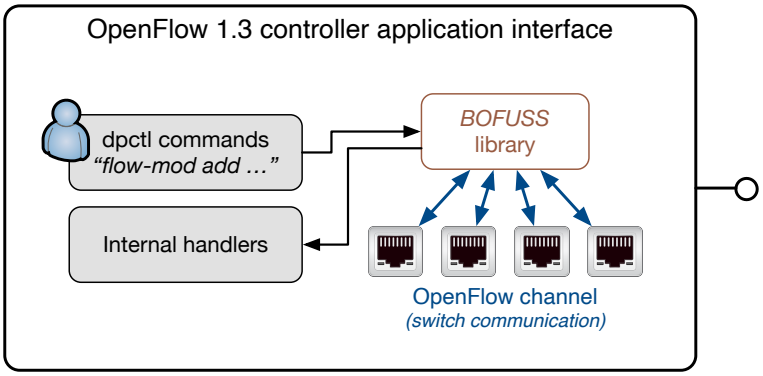

Fig. 3: The `OFSwitch13Controller` internal structure

For OpenFlow messages coming from the switches, the controller interface provides a collection of internal handlers to deal with the different types of messages. Some handlers cannot be modified by derived class, as they must behave as already implemented. Other handlers can be overridden to implement the desired control logic.

The *OFSwitch13* module brings the `OFSwitch13LearningController` class that implements the controller interface to work as a "learning bridge controller" (see 802.1D). This learning controller instructs the OpenFlow switches to forward incoming unicast frames from one port to the single correct output port whenever possible (similar to the `ns3::BridgeNetDevice`).

1.2.3 OpenFlow channel

The OpenFlow channel is the interface that connects switches to OpenFlow controllers. Through this interface, the controller configures and manages the switch. In the *OFSwitch13* module, the controller interface can manage the switch devices remotely over a separate dedicated network (out-of-band controller connection). It is possible to use standard *ns-3* protocol stack, channels and devices to create the OpenFlow channel connections using a single shared channel or individual links between the controller interface and each switch device. This model provides realistic control plane connections, including communication delay and, optionally, error models. It also simplifies the OpenFlow protocol analysis, as the *ns-3* tracing subsystem can be used for outputting PCAP files.

Considering that the OpenFlow messages traversing the OpenFlow channel follow the standard wire format, it is also possible to use the *ns-3* `TapBridge` module to integrate an external OpenFlow controller, running on the local machine, to the simulated environment.

1.2.4 BOFUSS library integration

This module was designed to work together with a OpenFlow user-space software switch implementation. The original Basic OpenFlow User Space Software Switch (BOFUSS) project (previously known as *ofsoftswitch13*) [Fernandes2020] was forked and modified for proper integration with *ns-3*, resulting in the BOFUSS library for OFSwitch13 library. The `master` branch does not modify the original switch datapath implementation, which is currently maintained in the original repository and regularly synced to this one. The modified `ns3lib` branch includes only the necessary files for building the *BOFUSS* library and integrating it with the *OFSwitch13* module.

The *BOFUSS* library provides the complete OpenFlow switch datapath implementation, including input and output ports, the flow-table pipeline for packet matching, the group table, and the meter table. It also provides support for converting internal messages to and from OpenFlow 1.3 wire format and delivers the `dpctl` utility for converting text commands into internal messages.

For proper *OFSwitch13* integration, the library was modified to receive and send packets directly to the *ns-3* environment. To this, all library functions related to sending and receiving packets over ports were annotated as *weak symbols*, allowing the *OFSwitch13* module to override them at link time. This same strategy was used for overriding time-related functions, ensuring time consistency between the library and the simulator. The integration also relies on *callbacks*, which are used by *BOFUSS* to notify the *OFSwitch13* module about internal packet events, like packets dropped by meter bands, packet content modifications by pipeline instructions, packets cloned by group actions, and buffered packets sent to the controller. As this integration involves callbacks and overridden functions, the module uses a global map to save pointers to all `OFSwitch13Devices` objects in the simulation, allowing faster object retrieve by datapath IP.

One potential performance drawback is the conversion between the *ns-3* packet representation and the serialized packet buffer used by the library. This problem is even more critical for empty packets, as *ns-3* provides optimized internal representation for them. To improve the performance, when a packet is sent to the library for pipeline processing, the module keeps track of its original *ns-3* packet using the `PipelinePacket` structure. For packets processed by the pipeline without content changes, the switch device forwards the original *ns-3* packet to the specified output port. In the face of content changes, the switch device creates a new *ns-3* packet with the modified content (discarding the original packet, eventually copying all packet and byte tags[1] to the new one). This approach is more expensive than the previous one but is far more simple than identifying which changes were made to the packet by the library.

1.3 Scope and Limitations

This module is intended for simulating OpenFlow networks, considering the main features available in OpenFlow version 1.3. The module provides a complete OpenFlow switch device and the OpenFlow controller interface. The switch is fully functional, while the controller interface is intended to allow users to write more sophisticated controllers to exploit the real benefits offered by SDN paradigm. However, some features are not yet supported:

- **Auxiliary connections**: Only a single connection between each switch and controller is available. According to the OpenFlow specifications, auxiliary connections could be created by the switch to improve the switch processing performance and exploit the parallelism of most switch implementations.
- **OpenFlow channel encryption**: The switch and controller may communicate through a TLS connection to provide authentication and encryption of the connection. However, as there is no straightforward TLS support on *ns-3*, the OpenFlow channel is implemented over a plain TCP connection, without encryption.

[1] Note that the byte tags in the new packet will cover the entire packet, regardless of the byte range in the original packet.

- **In-band control**: The OpenFlow controller manages the switches remotely over a separate dedicated network (out-of-band controller connection), as the switch port representing the switch's local networking stack and its management stack is not implemented.
- **Platform support**: This module is currently supported only for GNU/Linux platforms, as the code relies on an external library linked to the simulator that *must* be compiled with GCC.

1.4 *ns-3* OpenFlow comparison

Note that the *OFSwitch13* is not an extension of the available *ns-3* OpenFlow module. They share some design principles, like the use of an external software library linked to the simulator, the virtual TCAM, and the collection of `CsmaNetDevices` to work as OpenFlow ports. However, this is an entirely new code and can be used to simulate a broad number of scenarios in comparison to the available implementation.

One difference between the *ns-3* OpenFlow model and the *OFSwitch13* is the introduction of the OpenFlow channel, using *ns-3* devices and channels to provide the control connection between the controller and the switches. It allows the user to collect PCAP traces for this control channel, simplifying the analysis of OpenFlow messages. It is also possible the use of the *ns-3* `TapBridge` module to integrate a local external OpenFlow 1.3 controller to the simulated environment.

In respect to the controller, this module provides a more flexible interface. Instead of dealing with the internal library structures, the user can use simplified `dpctl` commands to build OpenFlow messages and send them to the switches. However, for processing OpenFlow messages received by the controller, the user still need to understand internal library structures and functions to extract the desired information.

In respect to the OpenFlow protocol implementation, the *OFSwitch13* module brings many improved features from version 1.3 in comparison to the available *ns-3* model (version 0.8.9). Some of the most important features are:

- **Multiple tables**: Prior versions of the OpenFlow specification did expose to the controller the abstraction of a single table. OpenFlow 1.1 introduces a more flexible pipeline with multiple tables. Packets are processed through the pipeline, they are matched and processed in the first table, and may be matched and processed in other subsequent tables.
- **Groups**: The new group abstraction enables OpenFlow to represent a set of ports as a single entity for forwarding packets. Different types of groups are provided to represent different abstractions such as multicasting or multipathing. Each group is composed of a set group buckets, and each group bucket contains the set of actions to be applied before forwarding to the port. Groups buckets can also forward to other groups.
- **Logical ports**: Prior versions of the OpenFlow specification assumed that all the ports of the OpenFlow switch were physical ports. This version of the specification adds support for logical ports, which can represent complex forwarding abstractions such as tunnels. In the *OFSwitch13* module, logical ports are implemented with the help of `VirtualNetDevice`, where the user can configure callbacks to handle packets properly.
- **Extensible match support**: Prior versions of the OpenFlow specification used a static fixed length structure to specify `ofp_match`, which prevents flexible expression of matches and prevents inclusion of new match fields. The `ofp_match` has been changed to a TLV structure, called OpenFlow Extensible Match (OXM), which dramatically increases flexibility.
- **IPv6 support**: Basic support for IPv6 match and header rewrite has been added, via the OXM match support.
- **Per-flow meters**: Per-flow meters can be attached to flow entries and can measure and control the rate of packets. One of the primary applications of per-flow meters is to rate limit packets sent to the controller.

For *ns-3* OpenFlow users who want to port existing code to this new module, please, check the *Porting ns-3 OpenFlow code* section for detailed instructions.

1.5 *ns-3* code compatibility

The only required modification to the *ns-3* source code for *OFSwitch13* integration is the inclusion of the new OpenFlow receive callback in the `CsmaNetDevice` and `VirtualNetDevice`. The module brings the patch for including this receive callback into *ns-3* source code, available under `utils/` directory.

The current *OFSwitch13* stable version is 5.2.2. This version is compatible with *ns-3* versions 3.38 and 3.39, and will not compile with older *ns-3* versions. If you need to use another *ns-3* release, you can check the RELEASE_NOTES file for previous *OFSwitch13* releases and their *ns-3* version compatibility, but keep in mind that old releases may have known bugs and an old API. It is strongly recommended to use the latest module version.

1.6 References

1. The reference [Chaves2016a] presents the *OFSwitch13* module, including details about module design and implementation. A case study scenario is also used to illustrate some of the available OpenFlow 1.3 module features.
2. The reference [Fernandes2020] describes the design, implementation, evolution and the current state of the *BOFUSS* project.
3. The references [Chaves2015], [Chaves2016b], and [Chaves2017] are related to the integration between OpenFlow and LTE technologies. The *ns-3* simulator, enhanced with the *OFSwitch13* module, was used as the performance evaluation tool for these works.

CHAPTER

TWO

USAGE

2.1 Building the Module

The *OFSwitch13* module interconnects the *ns-3* simulator and the *BOFUSS* software switch compiled as a library. Follow the instructions below to compile the *OFSwitch13* module. *Instructions were tested on Ubuntu 22.04.1 LTS.*[1]

2.1.1 Before starting

Before starting, ensure you have the following minimal requirements installed on your system:

```
$ sudo apt install g++ python3 cmake ninja-build git
$ sudo apt install make pkg-config libtool libboost-dev
```

2.1.2 Compiling the code

Clone the *ns-3* source code repository into your machine and checkout a stable version (we are using the ns-3.39):

```
$ git clone https://gitlab.com/nsnam/ns-3-dev.git
$ cd ns-3-dev
$ git checkout -b ns-3.39 ns-3.39
```

Download the *OFSwitch13* code into the `contrib/` folder.

```
$ cd contrib/
$ git clone https://github.com/ljerezchaves/ofswitch13.git
```

Update the *OFSwitch13* code to a stable version (we are using release 5.2.2, which is compatible with ns-3.39)[2]:

```
$ cd ofswitch13
$ git checkout 5.2.2
```

Go back to the *ns-3* root directory and patch the *ns-3* code with the appropriated `ofswitch13` patch available under the `ofswitch13/utils/` directory (check for the correct *ns-3* version):

```
$ cd ../../
$ patch -p1 < contrib/ofswitch13/utils/ofswitch13-3_39.patch
```

[1] Other distributions or versions may require different steps, especially regarding library compilation.

[2] Starting at *OFSwitch13* release 5.2.0, the `cmake` build system will automatically download and compile the correct version of *BOFUSS* library (Internet connection is required). For older *OFSwitch13* releases, we suggest you check the documentation and follow the proper build steps.

This patch creates the new OpenFlow receive callback at `CsmaNetDevice` and `VirtualNetDevice`, allowing OpenFlow switch to get raw packets from these devices. The module also brings a `csma-full-duplex` patch for improving CSMA connections with full-duplex support. This is an optional patch that can be applyed *after* the `ofswitch13` patch.

Now, configure the *ns-3*. By default, the `cmake` build system will handle *BOFUSS* library download and compilation. Anyway, if your want to use a custom *BOFUSS* library, use the `-DNS3_OFSWITCH13_BOFUSS_PATH` configuration option to specify its location:

```
$ ./ns3 configure
```

Check for the enabled *ns-3 OFSwitch13 integration* feature after configuration. Finally, compile the simulator:

```
$ ./ns3 build
```

That's it! Enjoy your *ns-3* fresh compilation with OpenFlow 1.3 capabilities.

2.2 Basic usage

Here is the minimal script that is necessary to simulate an OpenFlow 1.3 network domain (code extracted from `ofswitch13-first.cc` example). This script connects two hosts to a single OpenFlow switch using CSMA links, and configure both the switch and the controller using the `OFSwitch13InternalHelper` class.

```
#include <ns3/core-module.h>
#include <ns3/csma-module.h>
#include <ns3/internet-apps-module.h>
#include <ns3/internet-module.h>
#include <ns3/network-module.h>
#include <ns3/ofswitch13-module.h>

using namespace ns3;

int
main(int argc, char* argv[])
{
    // Enable checksum computations (required by OFSwitch13 module)
    GlobalValue::Bind("ChecksumEnabled", BooleanValue(true));

    // Create two host nodes
    NodeContainer hosts;
    hosts.Create(2);

    // Create the switch node
    Ptr<Node> switchNode = CreateObject<Node>();

    // Use the CsmaHelper to connect the host nodes to the switch.
    CsmaHelper csmaHelper;
    NetDeviceContainer hostDevices;
    NetDeviceContainer switchPorts;
    for (size_t i = 0; i < hosts.GetN(); i++)
    {
        NodeContainer pair(hosts.Get(i), switchNode);
        NetDeviceContainer link = csmaHelper.Install(pair);
        hostDevices.Add(link.Get(0));
```

(continues on next page)

(continued from previous page)

```
        switchPorts.Add(link.Get(1));
    }

    // Create the controller node
    Ptr<Node> controllerNode = CreateObject<Node>();

    // Configure the OpenFlow network domain
    Ptr<OFSwitch13InternalHelper> of13Helper = CreateObject<OFSwitch13InternalHelper>();
    of13Helper->InstallController(controllerNode);
    of13Helper->InstallSwitch(switchNode, switchPorts);
    of13Helper->CreateOpenFlowChannels();

    // Install the TCP/IP stack into hosts nodes
    InternetStackHelper internet;
    internet.Install(hosts);

    // Set IPv4 host addresses
    Ipv4AddressHelper ipv4Helper;
    Ipv4InterfaceContainer hostIpIfaces;
    ipv4Helper.SetBase("10.1.1.0", "255.255.255.0");
    hostIpIfaces = ipv4Helper.Assign(hostDevices);

    // Configure ping application between hosts
    PingHelper pingHelper(Ipv4Address(hostIpIfaces.GetAddress(1)));
    pingHelper.SetAttribute("VerboseMode", EnumValue(Ping::VerboseMode::VERBOSE));
    ApplicationContainer pingApps = pingHelper.Install(hosts.Get(0));
    pingApps.Start(Seconds(1));

    // Run the simulation
    Simulator::Stop(Seconds(10));
    Simulator::Run();
    Simulator::Destroy();
}
```

At first, don't forget to enable checksum computations, which are required by the *OFSwitch13* module. After creating host and switch nodes, the user is responsible for connect the hosts and switches to create the desired network topology. Using CSMA links for these connections is mandatory. Note that `CsmaNetDevices` created and installed into switch node will be later configured as switch ports. After connecting hosts and switches, it's time to create a controller node and configure the OpenFlow network. The `OFSwitch13InternalHelper` can be used to configure an OpenFlow network domain with internal controller application. The `InstallController()` method configures the controller node with a default OpenFlow learning controller application. The `InstallSwitch()` method installs the OpenFlow datapath into switch node and configures the switch ports. In the end, it's mandatory to call the `CreateOpenFlowChannels()` method to create the connections and start the communication between switches and controllers.

The rest of this example follows the standard *ns-3* usage: installing TCP/IP stack into host nodes, configuring IP addresses, installing applications and running the simulation. Don't install the TCP/IP stack into switches and controllers nodes (the helper does that for you). Also, don't assign an IP address to devices configured as switch ports. For instructions on how to compile and run simulation programs, please refer to the *ns-3* tutorial.

2.3 Helpers

2.3.1 OFSwitch13Helper

This module follows the pattern usage of standard helpers. The `OFSwitch13Helper` is a base class that must be extended to create and configure an OpenFlow 1.3 network domain, composed of one or more OpenFlow switches connected to single or multiple OpenFlow controllers. By default, the connections between switches and controllers are created using a single shared out-of-band CSMA channel, with IP addresses assigned to the 10.100.0.0/24 network. Users can modify this configuration by changing the `OFSwitch13Helper::ChannelType` attribute (dedicated out-of-band connections over CSMA or point-to-point channels are also available), or setting a different IP network address with the `OFSwitch13Helper::SetAddressBase()` static method. The use of standard *ns-3* channels and devices provides realistic connections with delay and error models.

This base class brings the methods for configuring the switches (derived classes configure the controllers). The `InstallSwitch()` method can be used to create and aggregate an `OFSwitch13Device` object to each switch node. By default, the `InstallSwitch()` method configures the switches without ports, so users must add the ports to the switch later, using the device `AddSwitchPort()`. However, it is possible to send to the `InstallSwitch()` method a container with `NetDevices` that will be configured as switch ports of a single switch node.

Each port is configured with the `CsmaNetDevice` created during the connection between switch nodes and other nodes in the simulation (the user must previously define these connections). It is also possible to use a `VirtualNetDevice` as a logical port, allowing the user to configure custom operations like tunneling.

After installing the switches and controllers, it is mandatory to use the `CreateOpenFlowChannels()` member method to effectively create and start the connections between all switches and all controllers on the same domain. After calling this method, you will not be allowed to install more switches nor controllers using this helper. Please note that this helper was designed to configure a single OpenFlow network domain. If you want to configure separated OpenFlow domains on your network topology (with their switches and controllers) so you may need to use a different helper instance for each domain.

This helper also allows users to enable some module outputs that are used for traffic monitoring and performance evaluation. Please, check the *Output* section for detailed information.

2.3.2 OFSwitch13InternalHelper

This helper extends the base class and can be instantiated to create and configure an OpenFlow 1.3 network domain composed of one or more OpenFlow switches connected to a single or multiple internal simulated OpenFlow controllers. It brings methods for installing the controller and creating the OpenFlow channels.

To configure the controller, the `InstallController()` method can be used to create a (default) new learning controller application and install it into the controller node indicated as parameter. It is also possible to install a different controller application other than the learning controller using this same method by setting the proper application parameter. Note that this helper is prepared to install a single controller application at each controller node, so don't install a second application on the same node, otherwise the helper will crash.

2.3.3 OFSwitch13ExternalHelper

This helper extends the base class and can be instantiated to create and configure an OpenFlow 1.3 network domain composed of one or more OpenFlow switches connected to a single external real OpenFlow controller. It brings methods for installing the controller node for TapBridge usage and creating the OpenFlow channels. The current implementation only supports the single shared CSMA channel type.

To configure the external controller, the `InstallExternalController()` method can be used to prepare the controller node so it can be used to connect internal simulated switches to an external OpenFlow controller running on the local machine over a TapBridge device. It installs the TCP/IP stack into controller node, attach it to the common CSMA channel, configure IP address for it and returns the `NetDevice` that the user will be responsible to bind to the TabBridge. Note that this helper is prepared to configure a single controller node. See the *External controller* section for details.

2.4 Attributes

2.4.1 OFSwitch13Controller

- `Port`: The port number on which the controller application listen for incoming packets. The default value is port 6653 (the official IANA port since 2013-07-18).

2.4.2 OFSwitch13Device

- `DatapathId`: The unique datapath identification of this OpenFlow switch. The datapath ID is a read-only attribute, automatically assigned by the object constructor.
- `CpuCapacity`: The data rate used to model the CPU processing capacity (throughput). Packets exceeding this capacity are discarded.
- `FlowTableSize`: The maximum number of entries allowed on each flow table.
- `GroupTableSize`: The maximum number of entries allowed on group table.
- `MeterTableSize`: The maximum number of entries allowed on meter table.
- `PipelineTables`: The number of pipeline flow tables.
- `PortList`: The list of ports available in this switch.
- `TcamDelay`: Average time to perform a TCAM operation in the pipeline. This value is used to calculate the average pipeline delay based on the number of flow entries in the tables.
- `TimeoutInterval`: The time between timeout operations in the pipeline. At each interval, the device checks if any flow in any table is timed out and update port status.

2.4.3 OFSwitch13Port

- `PortQueue`: The OpenFlow queue to use as the transmission queue in this port. When the port is constructed over a `CsmaNetDevice`, this queue is set for use in the underlying device. When the port is constructed over a `VirtualNetDevice`, this queue is not used.
- `QueueFactory`: The object factory describing the OpenFlow queue to be created for this port.

2.4.4 OFSwitch13Queue

- `QueueList`: The list of internal queues.

2.4.5 OFSwitch13PriorityQueue

- `QueueFactory`: The object factory describing the internal priority queues to be created.
- `NumQueues`: The number of internal priority queues.

2.4.6 OFSwitch13Helper

- `ChannelDataRate`: The data rate for the OpenFlow channel links.
- `ChannelType`: The type for the OpenFlow channel. Users can select between a single shared CSMA connection, or dedicated connection between the controller and each switch, using CSMA or point-to-point links.

2.4.7 OFSwitch13ExternalHelper

- `Port`: The port number on which the external controller application listen for incoming packets. The default value is port 6653 (the official IANA port since 2013-07-18).

2.4.8 OFSwitch13StatsCalculator

- `EwmaAlpha`: The EWMA alpha parameter, which is the weight given to the most recent measured value when updating average metrics values.
- `DumpTimeout`: The interval between successive dump operations.
- `OutputFilename`: The filename used to save OpenFlow switch datapath performance statistics.

2.5 Output

This module relies on the *ns-3* tracing subsystem for output. The `OFSwitch13Helper` base class allows users to monitor control-plane traffic by enabling PCAP and ASCII trace files for the `NetDevices` used to create the OpenFlow Channel(s). This approach can be useful to analyze the OpenFlow messages exchanged between switches and controllers on this network domain. To enable these traces, use the `EnableOpenFlowPcap()` and `EnableOpenFlowAscii()` helper member functions *after* configuring the switches and creating the OpenFlow channels. It is also possible to enable PCAP and ASCII trace files to monitor data-plane traffic on switch ports using the standard `CsmaHelper` trace functions.

For performance evaluation, the `OFSwitch13StatsCalculator` class can monitor statistics of an OpenFlow switch datapath. The instances of this class connect to a collection of trace sources in the switch device and periodically dumps the following datapath metrics on the output file:

1. [`LoaKbps`] CPU processing load in the last interval (Kbps);
2. [`LoaUsag`] Average CPU processing capacity usage (percent);
3. [`Packets`] Packets processed by the pipeline in the last interval;
4. [`DlyUsec`] EWMA pipeline lookup delay for packet processing (usecs);
5. [`LoaDrps`] Packets dropped by capacity overloaded in the last interval;

6. [MetDrps] Packets dropped by meter bands in the last interval;
7. [TabDrps] Unmatched packets dropped by flow tables in the last interval;
8. [FloMods] Flow-mod operations executed in the last interval;
9. [MetMods] Meter-mod operations executed in the last interval;
10. [GroMods] Group-mod operations executed in the last interval;
11. [PktsIn] Packets-in sent to the controller in the last interval;
12. [PktsOut] Packets-out received from the controller in the last interval;
13. [FloEntr] EWMA sum of entries in all pipeline flow tables;
14. [FloUsag] Average flow table usage, considering the sum of entries in all flow tables divided by the aggregated sizes of all flow tables with at least one flow entry installed (percent);
15. [MetEntr] EWMA number of entries in meter table;
16. [MetUsag] Average meter table usage (percent);
17. [GroEntr] EWMA number of entries in group table;
18. [GroUsag] Average group table usage (percent);
19. [BufPkts] EWMA number of packets in switch buffer;
20. [BufUsag] Average switch buffer usage (percent);

When the FlowTableDetails attribute is set to 'true', the EWMA number of entries and the average flow table usage for each pipeline flow table is also available under the columns `T**Entr` and `T**Usag`.

To enable performance monitoring, use the `EnableDatapathStats()` helper member function *after* configuring the switches and creating the OpenFlow channels. By default, statistics are dumped every second, but users can adjust this interval with the `OFSwitch13StatsCalculator::DumpTimeout` attribute. Besides, an Exponentially Weighted Moving Average (EWMA) is used to update the average values, and the attribute `OFSwitch13StatsCalculator::EwmaAlpha` can be adjusted to reflect the desired weight given to most recent measured values.

When necessary, it is also possible to enable the *BOFUSS* library logging mechanism using two different approaches:

1. The simplified `OFSwitch13Helper::EnableDatapathLogs()` static method dumps messages at debug level for all library internal modules into the output file (users can set the filename prefix);
2. The advanced `EnableBofussLog()` method allow users to define the target log facility (the console or a file), set the filename, and also customize the logging levels for different library internal modules.

2.6 Porting *ns-3* OpenFlow code

For *ns-3* OpenFlow users that want to port existing code to the new *OFSwitch13* module, keep in mind that this is not an extension of the available implementation. For simulation scenarios using the existing *ns-3* OpenFlow module configured with the `ns3::OpenFlowSwitchHelper` helper and using the `ns3::ofi::LearningController`, it is possible to port the code to the *OFSwitch13* module with little effort. The following code, based on the `openflow-switch.cc` example, is used for demonstration:

```
#include "ns3/openflow-module.h"

// Connecting the terminals to the switchNode using CSMA devices and channels.
// CsmaNetDevices created at the switchNode are in the switchDevices container.
// ...
```

(continues on next page)

(continued from previous page)

```
// Create the OpenFlow helper
OpenFlowSwitchHelper ofHelper;

// Create the learning controller app
Ptr<ns3::ofi::LearningController> controller;
controller = CreateObject<ns3::ofi::LearningController>();
if (!timeout.IsZero())
  {
    controller->SetAttribute("ExpirationTime", TimeValue(timeout));
  }

// Install the switch device, ports and set the controller
ofHelper.Install(switchNode, switchDevices, controller);

// Other configurations: TCP/IP stack, apps, monitors, etc.
// ...
```

This code creates an `ns3::ofi::LearningController` object instance as the controller. It also sets the internal attribute `ExpirationTime` for cache timeout. Then, the helper installs the OpenFlow switch device into the `switchNode` node. The CSMA devices from `switchDevices` container are installed as OpenFlow ports, and the `controller` object is set as the OpenFlow controller for the network. The following code implements the same logic in the *OFSwitch13* module:

```
#include "ns3/ofswitch13-module.h"

// Connecting the terminals to the switchNode using CSMA devices and channels.
// CsmaNetDevices created at the switchNode are in the switchDevices container.
// ...

// Create the OpenFlow 1.3 helper
Ptr<OFSwitch13InternalHelper> of13Helper = CreateObject<OFSwitch13InternalHelper>();

// Create the controller node and install the learning controller app into it
Ptr<Node> controllerNode = CreateObject<Node>();
of13Helper->InstallController(controllerNode);

// Install the switch device and ports.
of13Helper->InstallSwitch(switchNode, switchDevices);

// Create the OpenFlow channel connections.
of13Helper->CreateOpenFlowChannels();

// Other configurations: TCP/IP stack, apps, monitors, etc.
// ...

// Arbitrary simulation duration (can be changed for any value)
Simulator::Stop(Seconds(10));
```

Note that the *OFSwitch13* module requires a new node to install the controller application into it. The `InstallController()` function creates the learning application object instance and installs it in the `controllerNode`. Then, the `InstallSwitch()` function installs the OpenFlow device into `switchNode` and configures the CSMA devices from `switchDevices` container as OpenFlow ports. Finally, the `CreateOpenFlowChannels()` function configures the connection between the switch and the controller. Note that the `OFSwitch13LearningController` does not provide the

`ExpirationTime` attribute. Don't forget to include the `Simulator::Stop()` command to schedule the time delay until the Simulator should stop; otherwise, the simulation will never end.

For users who have implemented new controllers in the *ns-3* OpenFlow module, extending the `ns3::ofi::Controller` class, are encouraged to explore the examples and the Doxygen documentation for the `OFSwitch13Controller` base class. In a nutshell, the `ReceiveFromSwitch()` function is replaced by the internal handlers, used to process each type of OpenFlow message received from the switch. See the *Extending the controller* section for more details.

2.7 Advanced Usage

2.7.1 `dpctl` commands

For constructing OpenFlow messages and sending them to the switches, the controller relies on the `dpctl` utility to simplify the process. The `dpctl` is a management utility that enables some control over the OpenFlow switch. With this tool, it is possible to add flows to the flow table, query for switch features and status, and change other configurations. The `DpctlExecute()` function can be used by derived controllers to convert a variety of `dpctl` commands into OpenFlow messages and send it to the target switch. If the switch is not connected to the controller yet, this method will automatically schedule the commands for execution just after the handshake procedure between the controller and the switch. This is particularly useful for executing `dpctl` commands when creating the topology, before invoking `Simulator::Run()`.

Check the utility documentation for details on how to create the commands. Note that the documentation is intended for terminal usage in Unix systems, which is a little different from the usage in the `DpctlExecute()` function. For this module, ignore the options and switch reference, and consider only the command and the arguments. You can find some examples of this syntax at *The QoS controller example* source code.

2.7.2 Extending the controller

The `OFSwitch13Controller` base class provides the necessary interface for controller implementation. It uses the `dpctl` commands for sending OpenFlow messages to the switches. The controller also uses OpenFlow message handlers to process different OpenFlow message received from the switches. Some handler methods cannot be modified by derived class, as they must behave as already implemented. Other handlers can be overridden by derived controllers to proper parse packets sent from switch to controller and implement the desired control logic. The current implementation of these virtual handler methods does nothing: only free the received message and returns 0. Note that handlers *must* free received messages (msg) when everything is fine. For `HandleMultipartReply()` implementation, note that several types of multipart replies can be filtered.

In the `OFSwitch13LearningController` implementation, the `HandlePacketIn()` function is used to handle packet-in messages sent from switch to this controller. It looks for L2 switching information, updates the structures and sends a packet-out back to the switch. The `HandleFlowRemoved()` is used to handle expired flow entries notified by the switch to this controller. It looks for L2 switching information and removes associated entry.

The `QosController` example includes a non-trivial controller implementation that is used to configure the network described in *The QoS controller example* section. Several `dpctl` commands are used to configure the switches based on network topology and desired control logic, while the `HandlePacketIn()` is used to filter packets sent to the controller by the switch. Note that the *BOFUSS* function `oxm_match_lookup()` is used across the code to extract match information from the message received by the controller. For ARP messages, `HandleArpPacketIn()` exemplifies how to create a new packet at the controller and send to the network over a packet-out message. Developers are encouraged to study the library internal structures to understand better how the handlers are implemented and also how to build an OpenFlow message manually.

2.7.3 External controller

Considering that the OpenFlow messages traversing the OpenFlow channel follow the standard wire format, it is possible to use the *ns-3* `TapBridge` module to integrate an external OpenFlow 1.3 controller, running on the local system, to the simulated environment. The experimental `external-controller.cc` example uses the `OFSwitch13ExternalHelper` to this end, as follows:

```
// ...
// Configure the OpenFlow network domain using an external controller
Ptr<OFSwitch13ExternalHelper> of13Helper = CreateObject<OFSwitch13ExternalHelper>();
Ptr<NetDevice> ctrlDev = of13Helper->InstallExternalController(controllerNode);
of13Helper->InstallSwitch(switches.Get(0), switchPorts [0]);
of13Helper->InstallSwitch(switches.Get(1), switchPorts [1]);
of13Helper->CreateOpenFlowChannels();

// TapBridge the controller device to local machine
// The default configuration expects a controller on local port 6653
TapBridgeHelper tapBridge;
tapBridge.SetAttribute("Mode", StringValue("ConfigureLocal"));
tapBridge.SetAttribute("DeviceName", StringValue("ctrl"));
tapBridge.Install(controllerNode, ctrlDev);

// ...
```

The `InstallExternalController()` function configures the controller node as a "ghost node" on the simulator. This function returns the net device created at the controller node (`ctrlDev`), and the user is responsible for binding it to the `TapBridge` device, so it appears as if it were replacing the TAP device in the Linux. The default configuration expects that the OpenFlow controller is running on the local machine at port 6653 (the helper automatically sets the IP address). Users can modify the local port number setting the `OFSwitch13ExternalHelper::Port` attribute.

This example was tested with the Ryu controller (https://ryu-sdn.org) running on the local machine.

2.8 Examples

The examples are located in `examples/` folder.

2.8.1 Examples summary

- **ofswitch13-first**: Two hosts connected to a single OpenFlow switch. The default learning controller application manages the switch.
- **ofswitch13-multiple-controllers**: Two hosts connected to a single OpenFlow switch. The default learning controller application manages both switches.
- **ofswitch13-multiple-domains**: Two hosts connected to different OpenFlow switches. An independent default learning controller application manages each switch.
- **ofswitch13-single-domain**: Two hosts connected to different OpenFlow switches. Both switches are managed by the default learning controller application.
- **ofswitch13-custom-switch**: Two hosts connected to a single OpenFlow switch. The default learning controller application manages the switch. The switch datapath can be customized by the command line parameters.
- **ofswitch13-external-controller**: Two hosts connected to different OpenFlow switches. The same external controller application manages both switches.

- **ofswitch13-logical-port**: Two hosts connected to different OpenFlow switches. The tunnel controller application manages both switches. The ports interconnecting the switches are configured as logical ports, allowing switches to de/encapsulate IP traffic using the GTP/UDP/IP tunneling protocol.
- **ofswitch13-qos-controller**: It represents the internal network of an organization, where servers and client nodes are located far from each other. A specialized *OpenFlow QoS controller* manages the network, implementing some QoS functionalities and exploiting OpenFlow 1.3 features. *The QoS controller example* section below details this example.

2.8.2 The QoS controller example

A case study scenario was used by [Chaves2016a] to demonstrate how some of the available OpenFlow 1.3 module features can be employed to improve network management. Figure *Network topology for QoS controller example* shows the network topology used in this example. It represents the internal network of an organization, where servers and client nodes are located far from each other (e.g., in separated buildings). The "long-distance" connection between the sites is via two links of 10 Mbps each, while all the other local connections are 100 Mbps. On the server side, the *OpenFlow border switch* acts as a border router element: it is responsible for handling connection requests coming from the clients and redirecting them to the appropriate internal server. On the client side, the *OpenFlow client switch* is used to interconnect all clients in a star topology. Between these two switches, there is the *OpenFlow aggregation switch*, located at the border of the client side and used to provide improved long-distance communication. The default *OFSwitch13* learning controller is used to manage the client switch, whereas the new *OpenFlow QoS controller* is used to manage the other two switches. The latter controller implements some QoS functionalities exploiting OpenFlow 1.3 features, as described below. Each client opens a single TCP connection with one of the 2 available servers, and sends packets in uplink direction as much as possible, trying to fill the available bandwidth.

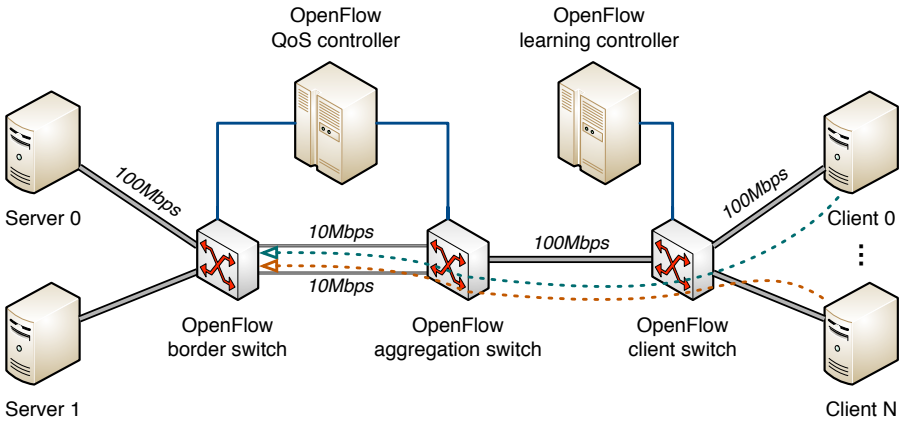

Fig. 1: Network topology for QoS controller example

Link aggregation: The link aggregation can be used to combine multiple network connections in parallel to increase throughput beyond what a single connection could sustain. The OpenFlow group table is used to split the traffic and implement the link aggregation.

OpenFlow groups were introduced in OpenFlow 1.1 as a way to perform more complex operations on packets that cannot be defined within a flow alone. Each group receives packets as input and performs any OpenFlow actions on these packets. The power of a group is that it contains separate lists of actions, and each action list is referred to as an OpenFlow bucket. There are different types of groups, and the *select* group type can be used to perform link aggregation. Each bucket in a select group has an assigned weight, and each packet that enters the group is sent to a single bucket. The bucket selection algorithm is undefined and is dependent on the switch's implementation (the *BOFUSS* library implements the weighted round robin algorithm).

In the proposed network topology, the QoS controller configures both the border and the aggregation switches to

perform link aggregation over the two narrowband long-distance connections, providing a 20 Mbps connection between servers and clients (use the `QosController::LinkAggregation` attribute to enable/disable this feature). Each OpenFlow bucket has the same weight in the select group, so the load is evenly distributed among the links.

Load balancing: A load balancing mechanism can be used to distribute workloads across multiple servers. Among many goals, it aims to optimize resource use and avoid overload of any single server. The most commonly used applications of load balancing is to provide single Internet service from multiple servers, also known as a server farm.

In the proposed network topology, the OpenFlow QoS controller configures the border switch to listen for new requests on the IP and port where external clients connect to access the servers. The switch forwards the new request to the controller, which decides which of the internal servers must take care of this connection. Then, it installs the match rules into border switch to forward the subsequent packets from the same connection directly to the chosen server. All this happen without the client ever knowing about the internal separation of functions.

To implement this load balancing mechanism, the QoS controller depends on the extensible match support introduced in OpenFlow 1.2. Prior versions of the OpenFlow specification used a static fixed length structure to specify matches, which prevents flexible expression of matches and prevents the inclusion of new match fields. The extensible match support allows the switch to match ARP request messages looking for the server IP address and redirect them to the controller, which creates the ARP reply message and send it back to the network. The set-field action is used by the border switch to rewrite packet headers, replacing source/destinations IP addresses for packets leaving/entering the server farm.

Per-flow meters: OpenFlow meter table, introduced in OpenFlow 1.3, enables the switch to implement various simple QoS operations. A meter measures the rate of packets assigned to it and enables controlling the rate of those packets. The meter triggers a meter band if the packet rate or byte rate passing through the meter exceeds a predefined threshold. If the meter band drops the packet, it is called a rate limiter.

To illustrate the meter table usage, the OpenFlow QoS controller can optionally limit each connection throughput to a predefined data rate threshold, installing meter rules at the border switch along with the load balancing flow entries (use the `QosController::EnableMeter` and `MeterRate` attributes to enable/disable this feature).

2.9 Troubleshooting

- If your simulation go into an infinite loop, check for the required `Simulator::Stop()` command to schedule the time delay until the Simulator should stop.
- Note that the `OFSwitch13LearningController` does not implement the Spanning Tree Protocol part of 802.1D. Therefore, it won't work if the network has loops on the connections between switches.
- For simulating scenarios with more than one OpenFlow network domain configured with the `OFSwtich13InternalHelper`, use a different helper instance for each domain.
- For using ASCII traces it is necessary to include the `ns3::PacketMetadata::Enable()` at the beginning of the program, before any packets being sent.

BIBLIOGRAPHY

[Chaves2015] Luciano J. Chaves, Vítor M. Eichemberger, Islene C. Garcia, and Edmundo R. M. Madeira. "Integrating OpenFlow to LTE: some issues toward Software-Defined Mobile Networks". In: 7th IFIP International Conference on New Technologies, Mobility and Security (NTMS), 2015.

[Chaves2016a] Luciano J. Chaves, Islene C. Garcia, and Edmundo R. M. Madeira. "OFSwitch13: Enhancing ns-3 with OpenFlow 1.3 support". In: 8th Workshop on ns-3 (WNS3), 2016.

[Chaves2016b] Luciano J. Chaves, Islene C. Garcia, and Edmundo R. M. Madeira. "OpenFlow-based Mechanisms for QoS in LTE Backhaul Networks". In: 21st IEEE Symposium on Computers and Communications (ISCC), 2016.

[Chaves2017] Luciano J. Chaves, Islene C. Garcia, and Edmundo R. M. Madeira. "An adaptive mechanism for LTE P-GW virtualization using SDN and NFV". In: 13th International Conference on Network and Service Management (CNSM), 2017.

[Fernandes2020] Eder L. Fernandes, Elisa Rojas, Joaquin Alvarez-Horcajo, Zoltan L. Kis, Davide Sanvito, Nicola Bonelli, Carmelo Cascone, and Christian E. Rothenberg. "The Road to BOFUSS: The Basic OpenFlow User-space Software Switch". Journal of Network and Computer Applications, 165:102685, 2020.