# A Software-Defined Spanning Tree Application for ns-3

Jared S. Ivey, Michael K. Riley, and George F. Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA, USA
{j.ivey, mriley7, riley}@gatech.edu

## 1. INTRODUCTION

This work demonstrates a spanning tree application for ns-3 under a software-defined networking (SDN) framework. SDN enables more flexibility in communication networks by separating the global routing decisions of the control plane from the simple packet forwarding of the data plane. Through standards such as the OpenFlow protocol, it simplifies the logical components directing the system by abstracting lower-level functionality. These concepts allow the network to be more easily partitioned, accommodating wider levels of research without interfering with typical end-user traffic flow. Furthermore, greater traceability is achieved due to the logical centralization of network control, enabling an infrastructure for more thorough network security measures.

To accommodate more scalable and resilient topologies in both traditional and software-defined networks, techniques must be employed to prevent extraneous behavior, such as forwarding or flooding loops. The basic spanning tree algorithm provides one method in SDN for preventing flooding loops that would otherwise be caused by simple packet-forwarding, learning applications. Unlike the traditional Spanning Tree Protocol, a spanning tree application in SDN offloads the responsibility of topology awareness and spanning tree creation to a logically centralized controller. The controller can then instruct the switches it controls to configure their ports based on the characteristics of the spanning tree. This work examines some of the visual and statistical effects of a spanning tree application on an n-by-n grid topology of switches in terms of latency in the basic network and reactivity to topology changes. NetAnim will be employed to visually demonstrate the correctness of the spanning tree effects.

## 2. BACKGROUND

### 2.1 OpenFlow Protocol

The OpenFlow communication protocol is a prevalent standard under which SDN may be deployed. It is an open protocol that enables researchers to run experimental protocols on large scale networks while maintaining the integrity of normal user traffic. With OpenFlow, the flow-tables contained in modern Ethernet switches and routers are simplified to accommodate a general set of functions and can be programmed according to these functions. An OpenFlow switch integrates a flow table, a secure channel, and the OpenFlow protocol. The flow table consists of a set of flow entries. A flow is a match qualifier linked with a list of actions to take if the specific match is found (possibly sending the packet out through a certain port, modifying some field or fields in the packet before forwarding it, or simply dropping the packet).

The secure channel of an OpenFlow switch connects it remotely to a process, referred to as the controller. Across this connection, the switch and controller can communicate commands and packets. This communication is standardized by the OpenFlow protocol which provides a means to interface with the switch without directly programming it. The controller can communicate appropriate actions for switches to take on packets by adding, modifying, or removing flow entries from the flow tables of the switch.

The formal OpenFlow protocol may be found in the *OpenFlow Switch Specification*. At the time of this work, versions extend from 1.0.0 to 1.5.0. This work and its implementation focuses primarily on version 1.0.0.

### 2.2 libfluid SDN Library

The **libfluid** library is actually a bundle of two separate libraries that provide the basic capabilities to implement an OpenFlow controller. The **libfluid_base** library provides the classes necessary to allow a formal OpenFlow connection, an OpenFlow controller to listen for these connections, and event handlers across the network. The **libfluid_msg** class library provides simple methods for creating, parsing, and formatting OpenFlow messages for transfer.

The **libfluid_base** library is designed according to a client-server architecture. `OFConnection` objects in the **libfluid_base** library encapsulate the attributes and components of OpenFlow connections. These objects maintain the connection state, OpenFlow version, and other attributes for a particular connection.

The **libfluid_msg** library provides a simplified interface for creating and analyzing OpenFlow messages. It provides the base class `OFMsg` that all other OpenFlow message objects inherit. Each child message class inherits two methods, `pack` and `unpack`. The `pack` method takes the contents of an `OFMsg` object and formats it in network byte order structures according to the appropriate OpenFlow specification. The `unpack` method performs the reverse operation.

### 2.3 ns-SDN

The design of the classes specific to providing SDN simulation capabilities in ns-3 primarily center on implementing an SDN controller and OpenFlow-enabled switch as user-defined applications. These applications in ns-3 are installed on nodes in the simulated topology where they may receive packets, perform a given set of actions based on the nature

of these received packets, and then forward them appropriately.

## 2.4 SDN Controller

The SDN controller application is composed of the `SdnListener` and `SdnController` classes while heavily relying on the `SdnConnection` class to communicate with the SDN switch application. The `SdnListener` class provides the basic capabilities to interface with a libfluid-style controller program. The `SdnController` class provides the underlying functionality to communicate the event-based instructions to the SDN switch application.

The `SdnListener` class provides the basic functionality handled in libfluid by an abstract controller example.This class can accept one of three types of controller events as an input to its `event_callback` method: `EVENT_SWITCH_UP`, `EVENT_SWITCH_DOWN`, `EVENT_PACKET_IN`, `EVENT_PORT_UP`, and `EVENT_PORT_DOWN`.

The `SdnConnection` class takes the place of the libfluid `OFConnection` class. This interface mimics that of the `OFConnection` well enough to reduce any compatibility issues between libfluid and ns-3. Underneath this interface, the `SdnConnection` class utilizes ns-3 sockets, providing communication capability suitable within an ns-3 simulation.

The `SdnController` class contains an `SdnListener` object to handle controller events, establishes connections to each switch to which it is directly connected, and handles packets that it receives from these connections. At startup, it will search each of its next hop connections to determine if an `SdnSwitch` application is installed on this subsequent node. If so, it will establish an `SdnConnection` with it. Once a connection has been established, the `SdnController` can call the `EVENT_SWITCH_UP` event on its `SdnListener`, allowing it to receive subsequent OpenFlow PacketIn messages that can prompt `EVENT_PACKET_IN` events on the `SdnListener`.

## 2.5 OpenFlow Switch

The SDN switch application is comprised of the `SdnPort`, `SdnFlowTable`, and `SdnSwitch` classes. `SdnPort` provides the formal definition of a binding port for the switch to send and receive data. The `SdnFlowTable` provides the structure and control for a table of flow rules for the switch to use on incoming packets. The `SdnSwitch` provides the actual application to act as a switch.

The `SdnPort` class is used as an enclosing class for switch-to-switch connections. When a switch needs to send out a packet, it does a lookup for the relevant `SdnPort`, grabs the relevant `SdnConnection`, and sends off the connection through the `NetDevice`.

The `SdnFlowTable` is responsible for all the flows a switch controls. `SdnFlowTable` is also responsible for keeping up with its own table statistics. The table also allows for adding, modifying, and deleting flow entries based on `SdnController` messages.

The `SdnSwitch` object is the main implementing class for an SDN-enabled switch. It functions as an ns-3 application installed on a node. The primary components of an `SdnSwitch` are an `SdnConnection` specifically connecting to an `SdnController` object, an `SdnFlowTable` that maintains the current flow rules to apply toward incoming packets, and a map of `SdnPort` objects to `SdnConnection` objects for all non-controller connections. The `SdnSwitch` objects receive data via callbacks from each given connection. At the `NetDevice` level (layer 2), packets can be retrieved with all of their headers still prepended exactly as they would have arrived to a real switch.

When an `SdnSwitch` handles data from a non-controller source, it sends the packet to the `SdnFlowTable`, returning the outPort from which the packet must be sent. As the packet was received at layer 2, it must also be sent from layer 2, sending out from the appropriate `NetDevice`. If a port of OFPP_NONE is returned from the table, the packet was not handled and must be sent to the controller via an OFPT_PACKET_IN message to request the appropriate action.

# 3. DEMONSTRATION

## 3.1 Spanning Tree Application

Spanning tree algorithms are implemented in typical network communication protocols to prevent forwarding loops by determining a single route between each pair of nodes within a topology. Within software-defined networking, the processing effort and topology awareness of the spanning tree algorithm can be offloaded to the controller as one of its applications. Within this application, the controller creates a flow for each of its switches to implement its own version of link-layer discovery (similar to the protocol of the same name), allowing it to determine the topology it controls. This topology is constructed as a set of enabled links between switches. The links are structures specifying the source and destination datapaths, the destination port, and the link delay. For the spanning tree, a current node is selected as the switch with the lowest datapath value. Its next-hop neighboring switches are then examined. If a destination switch has not previously been processed, the link to it will be set such that its port is allowed to flood packets. This setting is accomplished through a PortMod message with the `OFPPC_NO_FLOOD` bit unset. Then, each next-hop destination switch will selected as the current node. The process repeats until all switches in the tree have been examined. The controller can handle updates to the topology based on the entry or exit of switches. In this way, the controller can determine the difference between a previous topology and the current one and send new PortMod messages to the switches accordingly.

## 3.2 Simulated Topology

The simulated topology on which the spanning tree application will be employed is an n-by-n grid of switches (nodes with the `SdnSwitch` application installed) with each edge switch connected to a single host. Each switch is also directly connected to a single "controller" node (one which has the `SdnController` application installed). A slightly modified version of the `V4Ping` application predefined in ns-3 is installed on each host. The modifications produce a "ping-all" behavior that allows each host to send a specific number of pings to each other host in the topology. This behavior provides a mechanism for visually verifying the connectivity of the entire simulated topology. For other parts of the demonstration, the predefined `OnOffApplication` is installed on the hosts to produce streams of packets within the network. Some of the switches may then be "downed" by stopping their `SdnSwitch` applications. The reactivity of the network may be examined as the controller updates the spanning tree based on the new topology.