An OpenFlow Extension for the OMNeT++ INET Framework

Dominik Klein
University of Wuerzburg, Germany
dominik.klein@informatik.uni-wuerzburg.de

Michael Jarschel
University of Wuerzburg, Germany
michael.jarschel@informatik.uni-wuerzburg.de

ABSTRACT

Software Defined Networking (SDN) is a new paradigm for communication networks which separates the control plane from the data plane of forwarding elements. This way, SDN constitutes a flexible architecture that allows quick and easy configuration of network devices. This ability is particularly useful when networks have to be adapted to changing traffic volumes of different applications running on the network. OpenFlow is currently the most prominent approach which implements the SDN concept and offers a high flexibility in the routing of network flows.

In this paper, we describe the implementation of our model of the OpenFlow system in the INET framework for OMNeT++. We present performance results to show the correctness of our model. As a first application, we use the simulation model to assess the round-trip-times in a theoretical OpenFlow deployment in a real topology of a North-American Testbed.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol verification, Routing protocols*; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms

Design, Experimentation, Standardization, Verification

Keywords

OMNeT++, INET, OpenFlow, SDN

1. INTRODUCTION

The current Internet architecture struggles with several issues like missing routing scalability and mobility support as well as traffic engineering capabilities even for smaller domains. The underlying problem is the rigid and inflexible nature of the current Internet architecture. Missing common interfaces and heterogeneous hardware vendors make these issues even worse. Hence, there is a demand for a flexible architecture which also allows quick and easy configuration of network devices.

OMNeT++ 2013 March 5, Cannes, France.

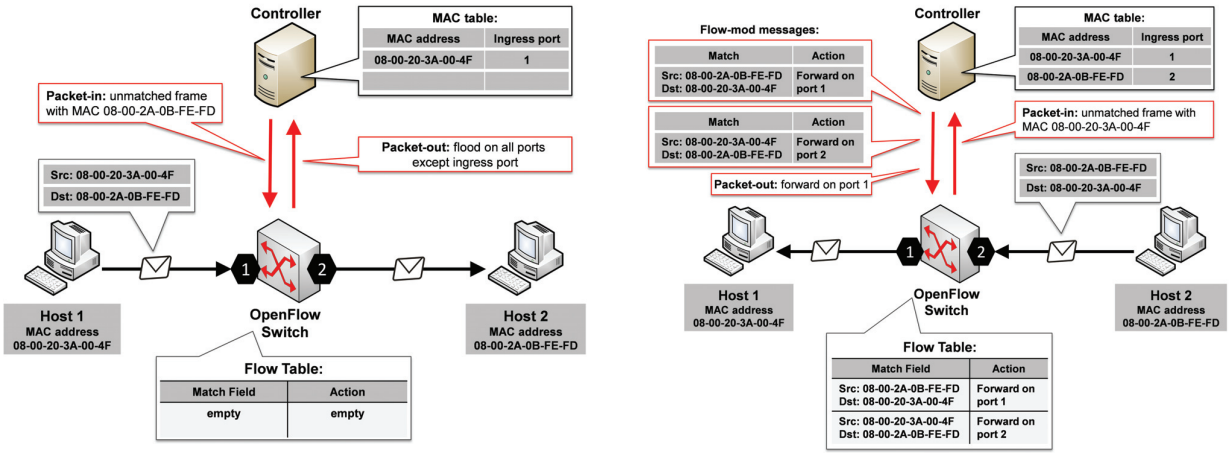
OpenFlow [6] is seen as one of the promising approaches to overcome the limitations of the current Internet stack. The development of the OpenFlow standard is managed by the Open Networking Foundation (ONF) [11]. The ONF promotes the development and use of Software Defined Networking (SDN) [10] technologies and OpenFlow is an example. OpenFlow offers a high flexibility in the routing of network flows and allows to change the behavior of a part of the network without influencing other traffic. This is achieved by separating the control plane in network switches from the data plane. A remote controller is responsible for all routing decisions and may change the forwarding rules of the switch.

OpenFlow and the first controller implementations are already deployed in networks. However, as the number and size of production networks deploying OpenFlow increases, relying on a single controller for the entire network might not be feasible for several reasons. The amount of control traffic directed towards the controller grows with the number of switches. In addition, the flow setup times increase, since the system is bounded by the processing power of the controller. So, there is the question what would be a better controller architecture with respect to performance, reliability, and scalability required to process new flows and program the switches.

There are several approaches for controller architectures with different advantages and disadvantages. So far, a single centralized controller is mainly used. But a network could also be split up in several parts and each part is managed by its own controller. Furthermore, a hierarchical structure is possible with a master controller that synchronizes all other controllers. In order to derive quantitative results, which are required for an appropriate dimensioning of the architecture with respect to the underlying physical network, a detailed performance evaluation framework is needed.

Until now, only few performance evaluations of OpenFlow architectures exist. Testbeds are often limited to smaller topologies, but the mentioned problems of a controller especially occur on larger, geographically distributed architectures. Simulation tools are much more suitable for this task since they allow to evaluate the scalability of the chosen controller. Furthermore, they allow to validate mechanisms before their deployment. Additionally, changes in the specification of the investigated technology can easily be integrated in the simulation model.

In this work, the OpenFlow components are integrated in the network simulation environment OMNeT++ [14]. Using the INET Framework [13], an OpenFlow switch and a basic controller are developed. For the controller, three possible behaviors are defined. In order to compare different controller architectures, a topology of 34 network nodes was created. The nodes represent cities across the USA which are part of a real network that is being established currently.



(a) First packet in flow from *host 1* to *host 2*.

(b) Response packet from *host 2* to *host 1*.

Figure 1: Communication between two nodes in an OpenFlow enabled network.

The remainder of this paper is structured as follows. In Section 2 we give a short overview of the related work. Section 3 provides an overview of OpenFlow and describes a simple communication scenario in an OpenFlow enabled network. Section 4 describes the implemented OpenFlow model and presents design choices. In Section 5, the performance of different controller architectures is evaluated for a real OpenFlow production network. Finally, Section 6 concludes this work.

2. RELATED WORK

An early approach towards simulating OpenFlow networks was made in 2009 with OpenFlowVMS [15]. This approach was based on using virtual machines to emulate OpenFlow-enabled devices. However, as virtual machines have a significant resource overhead, the system did not scale very well. Furthermore, it was designed for real-time functional OpenFlow testing rather than the simulation and evaluation of arbitrary scenarios. This is also true for Mininet, which is described by Lantz et al. in [5]. Mininet is a common tool to emulate network topologies for functional testing of OpenFlow controllers and applications. It abandons virtual machines in favor of lightweight namespaces to separate the emulated devices, thus reducing the overhead. The popular network simulator NS-3 also offers an OpenFlow simulation model [8]. However, this model in its current version does not model the OpenFlow controller as an external entity. Therefore, it is not possible to quantify the effects of the control channel or simulate multiple switches connected to a single OpenFlow controller.

3. OPENFLOW OVERVIEW

OpenFlow is an open standard that enables researchers to run experimental protocols in networks. It is added as a feature to commercial Ethernet switches, routers, or other network nodes. It provides a standardized hook to allow researchers to run experiments, e.g. to test and design new protocols, without requiring vendors to expose the internal workings of their network devices. In a classical router or switch design, the fast packet forwarding (data path) and the routing decisions (control path) occur on the same device. An OpenFlow switch separates these two functions. The data path portion still resides on the switch while routing decisions are moved to a separate controller, typically a standard server.

3.1 Connection Setup

Upon startup, the switch initiates a secure channel with the controller over TCP which is used by the controller to manage the switch via the OpenFlow protocol. The required IP address of the controller is known by configuration to the switch. It may also be possible that the controller has detected the switch and initiates the connection setup. Either way, once the secure TCP connection between switch and controller is established, both switch and controller exchange *hello* messages to learn the highest OpenFlow version which is supported by both entities. When an OpenFlow version supported by both is found, the controller requests the capabilities of the connecting switch via a *features request* message. The receiving switch answers the *features request* message with a *features reply* message and thereby informs the controller about its supported functionality.

3.2 Packet Processing

Upon reception of an Ethernet frame, an ordinary Ethernet switch usually looks up the destination MAC address in its MAC table which stores the mapping from MAC address to output switch port. This information is then used to forward the frame on the stored switch port. In addition, the mapping from source MAC address to ingress switch port is learned for incoming frames and stored in the MAC table.

In contrast, the data path of an OpenFlow switch contains one or more flow tables. Each flow table in the switch contains a set of flow entries where each entry contains match fields, counters, and instructions. An entry is identified by its extensible match fields which comprise the switch ingress port and different packet header fields. For received packets on the data path, the switch tries to match the ingress port and packet headers with the match fields in the different flow entries. The packet matching may span several tables and starts at fully defined entries and continues to less defined entries. To resolve possible draws of similar defined entries, a priority field indicates which entry should by chosen in that case.

The considered packet match fields in the flow table lookups depend on the packet type and typically include various packet header fields like for example the Ethernet source address or IPv4 destination address. A packet matches a flow table entry, if the values in the packet header fields utilized for the lookup match those defined in the flow table entry. If a flow entry field is wildcarded and has

a value of ANY, it matches all possible values in the header. Only the highest priority flow entry that matches the packet must be selected. The counters associated with the selected flow entry must be updated and the included instruction set must be applied.

If a matching entry is found, the instructions associated with the specific flow entry are executed. If no match is found in a flow table, the outcome depends on the switch configuration. The default in the OpenFlow switch specification version 1.2 [9] is to send packets to the controller over the OpenFlow channel via a *packet-in* message. Another option is to drop the packet. The *packet-in* message may either contain the entire packet or just a fraction of the packet header. In the second case, the packet is stored in a buffer at the switch and the *packet-in* message contains the buffer ID of the stored packet.

3.3 Controller Behavior

As explained in the prior section, the controller is connected via the secure channel to the switch and uses this channel to manage the flow entries in the flow table of the OpenFlow switch. In case the controller receives a *packet-in* message from the switch, it first examines the packet headers and checks whether a new flow entry needs to be created or what actions should be applied. If a new entry is required, the controller sends a *flow-mod* message to the switch which then installs the corresponding flow entry. In addition, the controller also sends a *packet-out* message which tells the switch to send the packet out of a specified port.

By specifying the OpenFlow protocol as a standard interface through which entries in the flow table can be defined externally, researchers can influence the switch behavior without the need to program the switch directly. The controller can add, update, and delete flow entries in flow tables by using the OpenFlow protocol.

3.4 Communication Example

The following example of an OpenFlow network consists of two hosts as well as an OpenFlow switch and an OpenFlow controller. The controller in this example implements ordinary learning switch behavior. The OpenFlow switch is connected to *host 1* via *port 1* and to *host 2* via *port 2*. For both hosts, the corresponding MAC addresses are shown. In Figure 1a, *host 1* with MAC address *08-00-20-3A-00-4F* wants to send a packet to *host 2* with MAC address *08-00-2A-0B-FE-FD*. In this example, the flow table of the OpenFlow switch is empty and so the switch does not know how to forward the packet. Hence, depending on the configuration, the switch either sends the entire packet or just the packet header to the connected controller. The controller decides about the actions that should be applied for this new flow and how this first packet is handled. In this case, the controller implements learning switch behavior and hence, the action in the *packet-out* message is flooding on all ports of the OpenFlow switch except the ingress port.

In addition, the mapping from source MAC address to ingress port is learned by the controller but no flow entry is installed in the switch as the destination MAC address has not yet been learned. Flow entries are first installed via *flow-mod* messages if the mappings for both addresses have been learned. This behavior is important because otherwise, the controller is unable to learn the source address in response packets as a flow entry with wildcarded source address prevents subsequent response packets from being sent to the controller.

The opposite direction for this communication is shown in Figure 1b. *Host 2* now sends a return packet to *host 1* which again arrives at the OpenFlow switch. As there is still no flow entry in the flow table, either the entire packet or the packet header is sent to the connected controller. The controller learns the mapping

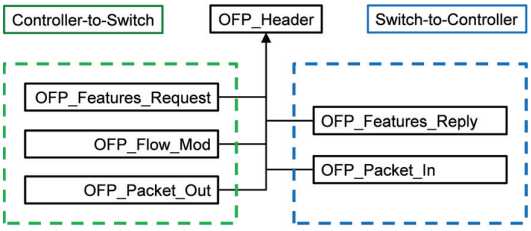


Figure 2: Implemented OpenFlow messages.

from source MAC address to ingress port and looks up the mapping for the destination MAC address. Now, both mappings are available and hence, the controller installs the corresponding flow entries via *flow-mod* messages in the flow table of the connected switch. In addition, the controller triggers the forwarding of the packet on *port 1* via a packet-out message to the connected switch. Subsequent packets of this flow can be sent without controller interactions because of the installed entries in the flow table of the switch.

4. SIMULATION MODEL

In this section, we describe our simulation model of the OpenFlow protocol as well as the implemented messages, network nodes, and utility modules.

4.1 Model Overview

Our model of the OpenFlow protocol is based on the OpenFlow switch specification version 1.2 [9] and uses the INET framework [13]. For the implementation of the model, we used the *openflow.h* header file to model the protocol and its defined messages as close as possible. The implemented nodes comprise the OpenFlow switch and OpenFlow controller as well as the most important messages which are required for the communication between switch and controller on the OpenFlow channel. In addition to the OpenFlow nodes, utility modules were implemented which provide further required functionality like a spanning tree module and a controller placement module. More details and chosen design options are presented in the following subsections.

4.2 OpenFlow Messages

The implemented messages of the OpenFlow protocol can be seen in Figure 2. Not all messages are implemented in the current state but the implemented subset nevertheless allows exhaustive simulations of OpenFlow enabled networks. All messages are sub-classes of the *OFP_Header* message class which includes the OpenFlow message header definition and the corresponding C++ structs so that the messages model their real counterparts as close as possible.

The implemented messages comprise the *OFP_Features_Request* and the *OFP_Features_Reply* which are required for the initialization of the OpenFlow channel between switch and controller. The *OFP_Packet_In* message is used by the switch to inform the controller about an unmatched incoming packet or to send an incoming packet to the controller if it is the associated action of a match. The message either contains the encapsulated packet or simply the buffer ID of the buffered packet. In the controller-to-switch direction, the *OFP_Packet_Out* message is used by the controller to send a packet out of a specified port at the switch. Finally, the controller can manage the flow table of the switch via the *OFP_Flow_Mod* message type which includes the packet match fields as well as the corresponding actions.

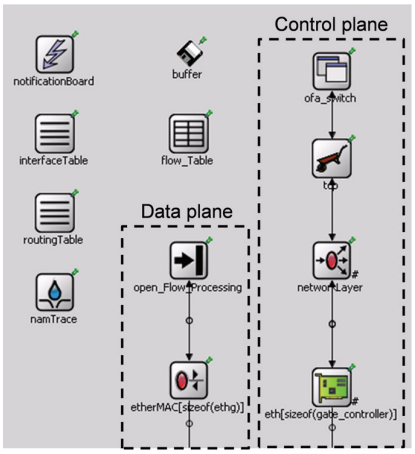


Figure 3: Implemented model of the OpenFlow switch.

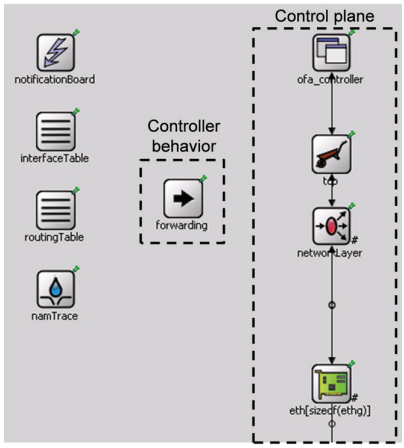


Figure 4: Implemented model of the OpenFlow controller.

4.3 OpenFlow Nodes

In this section, we give more implementation details and explain design choices for the two OpenFlow nodes.

4.3.1 OpenFlow Switch

The model of the OpenFlow switch can be seen in Figure 3. In the figure, we have highlighted the separation of data plane and control plane. The data plane contains an *EtherMAC* module which is connected to the outside and receives the incoming messages on the data plane. The received Ethernet frames are passed up without any modifications to the *OF_Processing* module which implements the OpenFlow switch functionality on the data plane. This functionality comprises flow table lookups for all incoming packets to find a possible match or informing the *OFA_Switch* module on the control plane about packet-in events of unmatched packets. As the applied operations may have different complexities, the *OF_Processing* module considers this fact by applying a service time which delays the simulation for the configured amount of time. This service time models the processing time of a real OpenFlow switches. In [4] for example, we have investigated this property for an OpenFlow switch and found out that the forward queue has an average service time of 9.8 µs.

For the inter-module communication between data plane and control plane, the OMNeT++ signal concept is used (see Section 4.14 in the OMNeT++ user manual). Hence, the *OF_Processing* module emits a *no-match-found* signal and the *OFA_Switch* module subscribes to this signal. The *OFA_Switch* module implements the OpenFlow switch functionality on the control plane and is responsible for the communication with the OpenFlow controller. Hence, on module startup, the *OFA_Switch* module establishes a TCP connection to the controller and negotiates the supported OpenFlow version and capabilities. In case of a *no-match-found* event, the *OFA_Switch* module takes the unmatched packet and prepares an *OFP_Packet_In* message for the controller. This message either contains the encapsulated complete packet or just the buffer ID. In case only the buffer ID is added, the *Buffer* module stores the packet until the *OFP_Packet_Out* message from the controller arrives. The option whether the packet is buffered or not can be configured via a configuration parameter of the switch.

The other modules in the control plane of the OpenFlow switch are part of the TCP/IP stack and are required as the *OFA_Switch* module is modeled as a TCP application because the OpenFlow channel uses a TCP connection. Over this channel, the *OFA_Switch* module receives *OFP_Packet_Out* and *OFP_Flow_Mod* messages from the controller. For each received *OFP_Flow_Mod* message from the connected controller, the *OFA_Switch* module takes the contained match and set of actions and adds a corresponding flow entry to the *FlowTable* module.

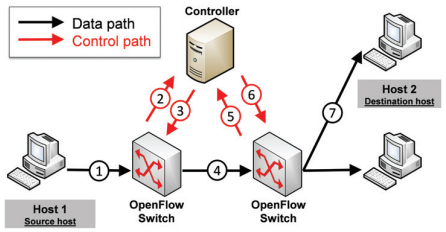
For received *OFP_Packet_Out* messages, the *OFA_Switch* module needs to trigger the *OF_Processing* module to apply the specified output action. This action could be either *flood-packet* or *send-packet*. For both actions, a corresponding event is emitted. Hence, the *OF_Processing* module is also subscribed to the associated *actions* signal. The emitted event may either contain the complete packet or just the buffer ID with which the packet can be retrieved from the *Buffer* module.

4.3.2 OpenFlow Controller

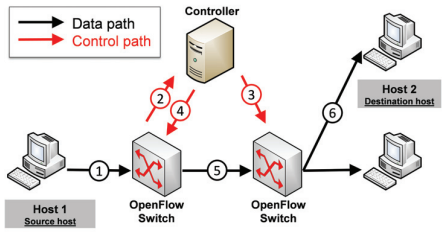
The model of the OpenFlow controller can be seen in Figure 4. The control plane part contains TCP/IP stack relevant modules as well as the *OFA_Controller* application module which realizes the controller functionality. Similar to the *OFA_Switch* module, the *OFA_Controller* module includes a service time parameter which considers the processing time of real OpenFlow controllers. According to [4], the average service time for the controller is set to 240 µs. However, this module only provides public methods which can be used to send *OFP_Packet_Out* and *OFP_Flow_Mod* messages to the connected OpenFlow switch. The actual controller behavior is realized in a separate module called *OF_Controller_App*. This is a controller module interface which is implemented by the different controller behavior modules. This way, it is very easy to configure as the desired behavior can be chosen via a configuration parameter.

The communication between the *OFA_Controller* module and the *OF_Controller_App* is again realized via the OMNeT++ signal concept. For each received *OFP_Packet_In* message, the *OFA_Controller* module emits a packet-in signal and includes the received packet. The *OF_Controller_App* module subscribes to this signal and takes the included packet. Other signaling messages like *OFP_Features_Reply* message are handled directly by the *OF_Controller* module which maintains a list of all connected OpenFlow switches and their supported OpenFlow version and switch capabilities. This stored information may be requested by the *OF_Controller_App* module, if the implemented function requires this knowledge about the connected switches.

As explained above, the *OF_Controller_App* module is a dummy



(a) Switch behavior.



(b) Forwarding behavior.

Figure 5: Comparison between *Switch* and *Forwarding* behavior for communication between source and destination host.

module which can be replaced by different controller behaviors and which receives packet-in signals from the *OF_Controller* module. Depending on its implemented behavior, the received signal induces different operations and once finished may trigger the required public methods at the *OFA_Controller* module. Currently, there are three implemented controller behaviors which are *Hub*, *Switch*, and *Forwarding* behavior.

Hub and *Switch* behavior modules model ordinary Ethernet hub and switch functionality and were implemented as a test for the OpenFlow protocol. A more sophisticated controller behavior is the *Forwarding* behavior. In the following, we explain the difference between *Switch* and *Forwarding* behavior by means of Figure 5 as this is required later on in Section 5.

Figures 5a and 5b both show the same network with three hosts and two OpenFlow switches interconnecting these hosts. Both OpenFlow switches are connected to the same OpenFlow controller. In Figure 5a, the data packet from *host 1* arrives at the first OpenFlow switch which has no flow entry yet and hence sends a *OFP_Packet_In* message to the controller. The controller answers with *OFP_Packet_Out* and *OFP_Flow_Mod* messages and the data packet is forwarded to the second switch which also has no flow entry yet. Hence, the signaling process between the OpenFlow switch and the OpenFlow controller is repeated. Eventually, the packet arrives at *host 2*. In Figure 5b, the same communication example is shown for an OpenFlow controller with *Forwarding* behavior. The forwarding behavior module has complete knowledge about the network and hence knows which other OpenFlow switches are located on the path to *host 2*. This knowledge is now used to signal the complete path from *host 1* to *host 2* in reverse order starting at the second switch. This way, the second OpenFlow switch already has the correct flow entry when the data packet arrives which reduces the total end-to-end delay and the number of signaling messages at the OpenFlow controller. In Section 5, two different controller behaviors for a real network topology will be compared.

4.4 Utility Modules

In this section, we describe two utility modules which have been implemented to ease the setup of larger networks. The first module ensures loop-free Ethernet networks by applying a spanning tree algorithm and the second module places an OpenFlow controller in a network at different locations.

4.4.1 *Spanning Tree*

For Ethernet switched networks where the underlying topology is not loop-free, the spanning tree protocol calculates loop-free spanning trees to avoid broadcast radiation where broadcast frames are for example repeatedly replicated in a loop of connected Ethernet switches. A similar functionality is implemented in the NOX controller [2] for OpenFlow networks. The basic spanning tree module builds a spanning tree but does not interact with the standard spanning tree protocol.

The spanning tree module in our model applies the same calculation but is realized in a separate module and is not part of the controller module. The spanning tree module has complete knowledge about the OpenFlow switch topology and builds the spanning tree at simulation start. Each OpenFlow switch interface on the data plane has a *NO_FLOOD* bit which is used for that purpose. If the bit is set, the corresponding link is not part of the spanning tree topology. The processing of received packets at OpenFlow switches hence needs to take the *NO_FLOOD* bit into account.

For each received packet on the data plane, the OpenFlow switch first identifies whether the ingress port of the data packet is part of the spanning tree (*NO_FLOOD* bit = false). In case the link is part of the tree, the packet is processed as usually. However in case the link is not part of the spanning tree, the packet is only further processed if the destination MAC or IP address is known. This way, packets which are broadcast on a link outside the spanning tree are not further processed at the next switch as the *NO_FLOOD* bit is set on the ingress port of that switch. A detailed description of the implemented spanning tree algorithm can be found in [1].

4.4.2 *Controller Placement*

The motivation for the controller placement module is to evaluate the effect of different controller placements on a variety of performance metrics, for example the mean round-trip-time for the hosts in the investigated network. In the current version, only a single OpenFlow controller is dynamically placed in the network and directly connected via a separate link to the different OpenFlow switches. This approach can be compared with out-of-band signaling where signaling messages are transmitted via a separate management connection. To account for different controller placements, the link delay between switch and controller is set according to the shortest path on the data plane.

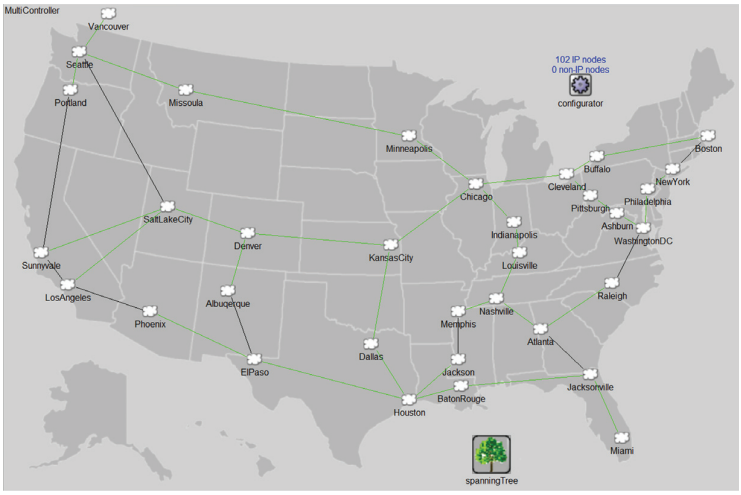
5. EVALUATION

In this section, we present a brief evaluation with respect to the mean round-trip-time in an OpenFlow enabled network for two different controller architectures.

5.1 Simulation Setup

In [3], different controller placements were evaluated for the Open Science, Scholarship, and Services Exchange (OS^3E) infrastructure topology [7]. The OS^3E infrastructure is one of the first production deployments of OpenFlow technology and hence it is an ideal reference scenario. Our model of this topology can be seen in Figure 6a and we conduct a similar evaluation as in [3] to prove the correctness of our implemented model.

The topology comprises *34* locations which are represented by OpenFlow domains. Each domain consist of an OpenFlow switch, several hosts and optionally an OpenFlow controller. The number of hosts can be defined by the user. The different domains are con-



(a) OMNeT++ network model.

Index	Location	Index	Location
1	Vancouver	18	Louisville
2	Seattle	19	Nashville
3	Portland	20	Memphis
4	Sunnyvale	21	Jackson
5	Los Angeles	22	Baton Rouge
6	Missoula	23	Cleveland
7	Salt Lake City	24	Pittsburgh
8	Phoenix	25	Atlanta
9	Denver	26	Jacksonville
10	Albuquerque	27	Buffalo
11	El Paso	28	Ashburn
12	Minneapolis	29	Raleigh
13	Kansas City	30	Washington DC
14	Dallas	31	Miami
15	Houston	32	Philadelphia
16	Chicago	33	New York
17	Indianapolis	34	Boston

(b) Domain indices

Figure 6: Simulation model and domain indices for OS^3E infrastructure topology

nected via a new channel type which uses the geographical distance between two locations to calculate the delay on that link. The link delay for fiber cable can be calculated as the distance divided by the propagation speed for optical fiber. The propagation speed is defined as the speed of light divided by the refraction index which is *1.5* for fiber cable. Hence, the link delay is obtained by dividing the geographical distance by two thirds of the speed of light.

As the performance metric for our evaluation is the mean round-trip-time (RTT), each host contains a modified ping application. For each sent echo request message, the application chooses a random host among the available *34* domains and then measures the RTT for the received echo reply message. After a ping has been performed, the application waits two seconds and then starts the next ping to another random destination. The simulation time for each run is set to *500 seconds* which results in about *250 RTT* measurements per domain. Over these 250 measurements, we compute the *mean RTT per domain* in seconds which indicates, how well the domain is connected to other domains in terms of RTT. This is done for all domains per run which results in one mean RTT value per domain and 34 mean RTT values in total. In addition, we repeat each run eight times with different random number seeds in order to exclude simulation artifacts and to obtain mean and confidence intervals over the different repetitions.

5.2 Multiple Controllers

In the first scenario, each location has its own OpenFlow controller which implements Ethernet switch behavior. The OpenFlow switches are configured to buffer the packets and send the header fields to the connected controller. The entire network uses Ethernet links and there is one endhost per OpenFlow domain.

As the evaluated topology is not loop-free, the *spanning tree* module needs to be placed in the network. As an example, the green links in Figure 6a show the calculated spanning tree for *KansasCity* as root. When choosing the root, care has to be taken because a bad spanning tree may negatively influence the mean round-trip-time for a certain location as data packets only take the links on the spanning tree which are not necessarily part of the shortest path. At simulation start, the controllers with switch behavior have not yet learned any mappings from MAC address to port and hence, all packets are broadcast along the links of the spanning tree. This way however, the controllers only learn ports which correspond to the spanning tree links possibly resulting in much longer paths. In Figure 6a for example, *Jacksonville* and *Atlanta* are directly connected but this link is not part of the spanning tree. Hence, data packets take a large detour via among others *Houston*, *KansasCity*, *Chicago*, and *Nashville*.

To further investigate that effect, we conduct the mean round-trip-time evaluation for each location as root of the spanning tree and hence obtain *34* different runs. Figure 7 shows the results as boxplot. The x axis shows the spanning tree root location index (see Table in Figure 6b) and the y axis shows the mean RTT per domain. The indexing of the OpenFlow domains starts at Vancouver in the northwest and continues to Boston in the northeast. Hence, domains with an index lower than 11 are located in the western part of the map while domains with an index larger or equal 11 are located in the eastern part.

For each spanning tree root location, the central red line in the blue box denotes the median over all mean RTTs while the edges are the 25th and 75th percentiles. The whiskers extend to extreme outliers but are at most 1.5 of the inter-percentile difference. Data points which lie outside the whisker range are drawn as red crosses. The boxplot representation gives a nice understanding about how the mean RTT values are distributed for a specific spanning tree. For a good spanning tree, not only the median should be small but also the mean RTT values should be relatively dense around the median which shows that each domain has a similar mean RTT. The pure median would not be sufficient because it does not consider extreme outliers. The spanning trees which are for example rooted at location 13 (Kansas City) and 16 (Chicago) offer similar median and percentiles but Chicago has the larger whisker range which denotes domains with a very high mean RTT. Considering that, the spanning tree which is rooted at Kansas City offers the best mean RTT over all domains. The worst spanning tree is rooted either at location 1 (Vancouver) or location 2 (Seattle) and has the highest median and the most extreme outliers. For that tree, all paths from southern domains to northern domains go through Seattle which results in a very high mean RTT.

The presented *multiple controller* scenario should mainly demonstrate that our Ethernet switch implementation inside the OpenFlow controller works and that the entire network behaves like an Ether-

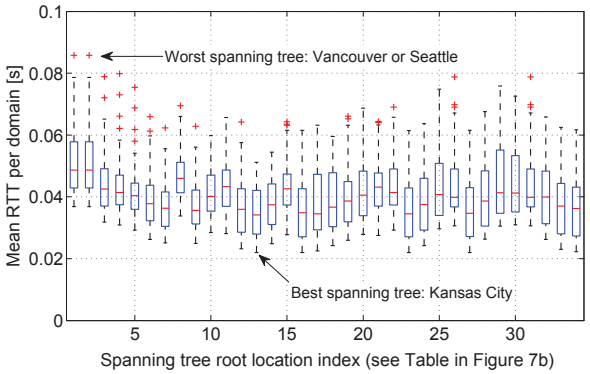


Figure 7: Boxplot of mean RTT per domain for all 34 different spanning tree roots.

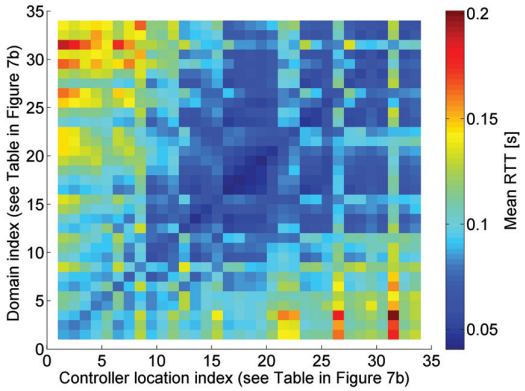


Figure 8: Surface plot of mean RTT per domain for different controller placements.

net network. A typical OpenFlow deployment however would not comprise an OpenFlow controller per domain but usually would involve only a few controllers which allow a central management of the entire network. Considering that, we present another extreme scenario with only one OpenFlow controller for the entire network in the next section.

5.3 Single Controller

In the second scenario, there is only one central OpenFlow controller which is connected via separate control links to all possible 34 domains. We use the *controller placement* module (see Section 4.4.2) to simulate different placements of the OpenFlow controller by setting the control link delay according to the theoretical placement of the OpenFlow controller. This way, we want to investigate what would be the best location for the controller with respect to mean RTT of the different domains.

The central controller in this scenario has complete knowledge about the network and uses the *Forwarding* behavior module as explained in Figure 5b for the set up of flow entries. The spanning tree module is also required as the underlying Ethernet topology remains the same as in the first evaluation. However, the spanning tree has less influence as the *Forwarding* behavior module calculates and installs the shortest path towards the destination MAC address in all OpenFlow switches on the shortest path towards the destination. Only the initial ARP request messages are broadcast and traverse the spanning tree. Subsequent echo request and echo reply message are unicast and follow the installed shortest path.

To get a general feeling about the effect of the different controller placements, Figure 8 shows a graphical representation of the different mean RTT values per domain and per controller placement. The x axis shows the different possible controller placements represented by the index of the domain in which the controller has been placed. The y axis shows the different domains for which the mean RTT has been measured to all other domains. The color of the rectangle denotes the value of the mean RTT according to the color bar shown on the right hand side of the figure. Blue colors denote a low mean RTT while red colors denote a high mean RTT value. The green rectangle at *(8,1)* shows for example that if the controller is placed in domain 8 (Phoenix), domain 1 (Vancouver) has a medium mean RTT of about 0.12 s to all other domains.

The brighter areas in the lower right and upper left corner of the plot constitute measurements with higher mean RTT values and can be explained due to the large distance between domain and controller location in those cases. The distance between switch and controller is highest if the controller is for example located in the western part while the switch is located in the eastern part or vice versa. This is true when the controller index is low and the domain index is high or the controller index is high and the domain index is low. The two red spots in the lower right corner for example correspond to controller placements in Jacksonville (*index 26*) and Miami (*index 31*). For those placements, the three domains located in the northwest (*Vancouver, Seattle, and Portland with index 0, 1, and 2*) have a great distance to the controller and hence, the mean RTT is larger than 0.15 s. The large dark blue area from the center to the upper right corner corresponds to controller placements and domain indices larger than 11. Domains with index larger than 11 are in the eastern part of the network where the topology is dense and well connected. Placing the controller there results in a very low mean RTT for those domains and in a moderate mean RTT for the domains in the western part except for the two controller placements in Jacksonville (*index 26*) and Miami (*index 31*).

The presentation in Figure 8 gives a nice understanding about the correlation between controller placement and mean RTT per domain but it is difficult to see which placement is the best. For that purpose, Figure 9 shows the results again as boxplot. The x axis shows the controller location index (see Table 6b) and the y axis shows the mean RTT per domain. For each controller location, the central red line in the blue box denotes the median over all mean RTTs while the edges are the 25th and 75th percentiles. The whiskers extend to extreme outliers but are at most 1.5 of the inter percentile difference. Data points which even lie outside the whisker range are drawn as red crosses.

It is apparent that the controller placements in the core of the topology (*indices 16 till 19*) provide the best results with respect to the mean RTT per domain. From these four possibilities, the placement in Nashville with *index 19* has the lowest median mean RTT while the placement in Louisville with *index 18* has a somewhat higher median but the whisker range is smaller. So the placement choice depends on whether the median should be smaller or the worst case mean RTT should be smaller. These results differ from those in [3] but can be explained due to not considered effects like the spanning tree and a slightly different performance metric. Our evaluation involves the simulation of an Ethernet network with all comprised effects and as performance metric, we use the mean RTT of a single domain to all other domains. The authors in [3] solve a theoretical facility location problem and hence consider the mean latency between controller and the different domains as metric. Both approaches are suitable while ours offers more flexibil-

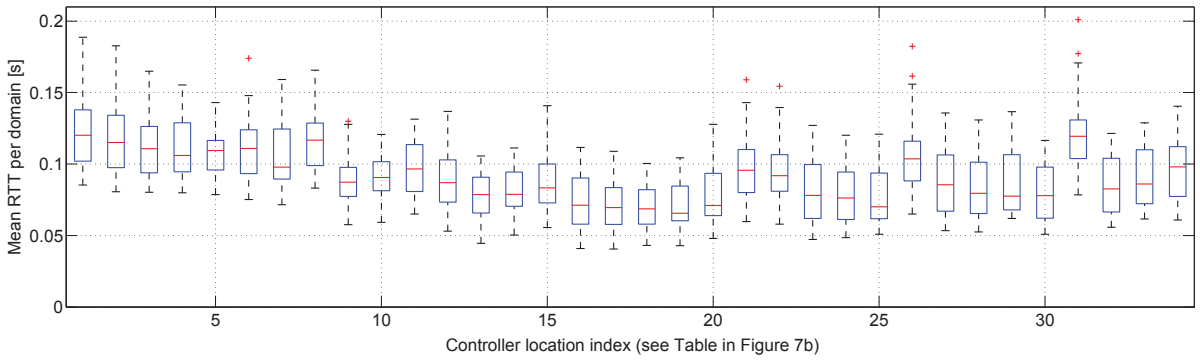


Figure 9: Boxplot of mean RTT per domain for different controller placements.

ity for example with respect to different realizations of distributed controller architectures. We are currently working on that feature so not only single controller scenarios can be evaluated.

6. CONCLUSION

In this work, we presented the integration of the OpenFlow protocol version 1.2 [9] in the INET framework for OMNeT++. The implemented simulation model comprises OpenFlow switch, OpenFlow controller, and the most important messages required to simulate OpenFlow enabled networks. Whenever possible, we used the *openflow.h* header file to model the protocol and its messages as close as possible.

As first evaluation scenarios, we evaluated different controller architectures for the Open Science, Scholarship, and Services Exchange (OS^3E) infrastructure topology which is one of the first OpenFlow productive networks. The presented *single controller* scenario demonstrates for example what is possible with our implementation of the OpenFlow protocol in INET. However, the currently implemented OpenFlow model is restricted to single controller deployments and distributed controller architectures are not possible in the current state.

As future work, we will extend our model so that it can be used as generic framework for the simulation of more sophisticated distributed controller architectures, which use for example a hierarchical approach. Therefore, it is necessary to introduce different controller domains which partition the network. This approach further requires inter-controller communication and a master controller which manages the different domains and ensures connectivity. As a last step, also resilience should be considered, for example by introducing backup controllers. The current implementation is available at [12] so that other researchers may use and extend our OpenFlow simulation framework.

7. ACKNOWLEDGEMENTS

The authors would like to thank Christian Rachor for the implementation as well as Rastin Pries and Prof. Tran-Gia for the fruitful discussion and the support in this work.

8. REFERENCES

[1] G. Gibb. Basic Spanning Tree for NOX Controller. http://www.openflow.org/wk/index.php/Basic_Spanning_Tree, Nov 2010.

[2] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, Jul 2008.

[3] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 7–12, New York, NY, USA, 2012. ACM.

[4] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia. Modeling and Performance Evaluation of an OpenFlow Architecture. In *23rd International Teletraffic Congress (ITC 2011)*, San Francisco, CA, USA, Sep 2011.

[5] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar 2008.

[7] Network Development and Deployment Initiative (NDDI). Open Science, Scholarship, and Services Exchange (OS^3E). http://www.internet2.edu/network/ose/, 2012.

[8] NS-3 v3.16. OpenFlow switch support. http://www.nsnam.org/docs/release/3.16/models/html/openflow-switch.html, Dec 2012.

[9] Open Networking Foundation. OpenFlow Switch Specification - Version 1.2 (Wire Protocol 0x03). https://www.opennetworking.org/images/stories/downloads/specification/openflow-spec-v1.2.pdf, Dec 2011.

[10] Open Networking Foundation. Software-Defined Networking: The New Norm for Networks. In *ONF White Paper*, Apr 2012.

[11] D. Pitt. Open Networking Foundation. https://www.opennetworking.org/, 2012.

[12] University of Wuerzburg. Openflow research activities. http://www3.informatik.uni-wuerzburg.de/research/ngn/openflow.shtml, Feb 2013.

[13] A. Varga. INET Framework for the OMNeT++ Discrete Event Simulator. http://github.com/inet-framework/inet, 2012.

[14] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *International Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, Mar 2008.

[15] K.-K. Yap. OpenFlowVMS - Simulating OpenFlow Network(s). http://www.openflow.org/wk/index.php/OpenFlowVMS, Mar 2011.