Simulation Framework for Distributed SDN-Controller Architectures in OMNeT++

Nicholas Gray, Thomas Zinner, Steffen Gebert, Phuoc Tran-Gia

University of Würzburg, Institute of Computer Science,
Am Hubland, 97074 Würzburg, Germany
{nicholas.gray,zinner,steffen.gebert,trangia}@informatik.uni-wuerzburg.de

Abstract. SDN introduces the separation of network control and network data plane. The control plane is removed from distributed network entities and logically centralized as the SDN controller. To provide resilience and performance such a logically centralized controller may again be physically distributed. Scenarios featuring distributed controller architectures include data center deployments, where controller instances synchronize states on small distances and delays, or continental WAN deployments with long distances and delays between controllers. The contribution of this paper is an OMNeT++ based simulation framework for assessing the performance of distributed SDN controller architectures. Relevant protocols and controller applications are modelled with a high level of detail. Further, an exemplary implementation of two different controller architectures, namely Hyperflow and Kandoo, is included. Initial results based on the provided implementations are presented.

1 Introduction

Software Defined Networking (SDN) [11] promotes the separation of the control and data plane in communication networks. While the data plane is kept distributed, the control plane is logically centralized in a controller and serves as interface for configuration purposes. Devices in the data plane are controlled via the southbound API using communication protocols like OpenFlow [15]. Thus, SDN provides a higher configuration flexibility and enables the network operator to dynamically react to changing network parameters. This results in a more efficient resource utilization as well as in a reduction of the management efforts. Hence, SDN is enjoying an increasing popularity and is already deployed in live production environments, i.e. Google's B4 backbone [9].

Despite the advantages of a centralized control plane, it also imposes new challenges in terms of resiliency and scalability limitations. Today's data centres, for example, are required to handle 150 million flows per second, while current OpenFlow controllers only have the ability to process a fraction of this demand [22]. One possible approach to address the scalability issues is to distribute the load among several controller instances while keeping the management logically centralized. In this context a variety of distributed controller architectures have been proposed, e.g., Hyperflow or Kandoo. The impact of different controller

architectures on relevant performance metrics like latencies or the resulting architectural overhead have not been in the focus of scientific investigations yet.

The contribution of this work is the design and implementation of a flexible simulation model capable of evaluating distributed SDN controller architectures within the OMNeT++ framework. Furthermore, we present an exemplary evaluation demonstrating the capabilities of the implemented framework by analysing the impact of distributed controller architectures on the offered traffic of the individual controllers.

The remainder of this work is structured as follows. Section 2 summarizes relevant background and related work on SDN, controller architectures, OMNeT++ and the performance evaluation of SDN. The implemented framework is introduced in Section 3. The impact of physically distributing the control plane on the offered control plane traffic is illustrated in Section 4. The paper is concluded in Section 5.

2 Background

This section summarizes relevant background and related work on SDN, OMNeT++, as well as controller architectures and their performance evaluation.

2.1 SDN & OpenFlow

The concept of Software-defined networking (SDN) has been introduced to achieve a higher configuration flexibility as well as a reduction in the complexity of modern network architectures. To accomplish this task, SDN is driven by four core principles, i.e., separation of control and data plane, a logically centralized controller, open standards and a programmable interface [11].

The communication between the SDN controller and the switch, which reflects the separation of the control and data plane is defined by a particular protocol i.e., OpenFlow. Whereas the SDN enabled switch is often a dedicated networking device, the SDN controller is implemented as software and is usually deployed on a standard server component.

Whenever an OpenFlow enabled switch receives a packet from the data plane it first performs a lookup in its flow table to determine if a matching rule is already existent. In the case no rule is found the switch extracts the header fields of the packet and creates a Packet-in message, which is sent to the controller asking for further instructions on how to handle the received packet. Once the Packet-in message is received and processed by the SDN controller it responds with a Packet-out message, which details how the switch should forward the packet. In addition the controller may send a Flow-mod message, which installs a rule into the switch's flow table and handles future matching packet without controller intervention.

2.2 OMNeT++ & INET

OMNeT++ [16] is a discrete event based network simulation framework which follows a component based design pattern and therefore makes it easy to be extended by third party modules. At its core, the OMNeT++ network simulation framework features the concept of *simple* and *compound* modules, which represent the individual entities of a simulation. To enable the exchange of information during the simulation, the modules need to encode the relevant data into Messages, which are then sent through gates to other connecting modules. The INET Framework [5] is a third party extension to OMNeT++ and provides the most common protocols used in the Internet. In particular, protocols such as TCP, UDP, IPv4, IPv6, FTP and many more have been implemented. In addition, INET also provides simple applications and entities, which make use of the underlying protocols and which can be reused within custom simulations.

2.3 Distributed Controller Architectures

To mitigate the drawbacks of the single point of failure and scalability limitations imposed by the SDN controller, the logically centralized control plane has to be physically distributed. This results in fundamental trade-offs like staleness of information vs. optimality of decisions [14]. Distributed controller algorithms, e.g., following a horizontal or a hierarchical architecture, allow to control this trade-off by adjusting their synchronization mechanism.

Relevant horizontally distributed controller implementations are HyperFlow [21] and OpenDaylight [17]. Whereas OpenDaylight uses a DHT which is accessible to all cluster nodes, HyperFlow stores state changing events to a distributed file system. Kandoo [7] follows the hierarchical approach by introducing local and root controllers. Local controllers handle all requests for which their local domain knowledge is sufficient and forward requests to the root controller, which require the global view of the network. In addition to Kandoo, the ONOS [2] controller also incorporates a hierarchical design. Onix [13] and Disco [18] differentiate between an intra and inter domain context and choose the method of synchronization accordingly. In this context, both controllers utilize a horizontal architecture for intra domain communication in form of a distributed data store and for inter domain purposes a hierarchical organization is created by only exposing aggregated information of an individual domain.

2.4 Performance Evaluation of SDN Architectures

One possibility to investigate the performance of distributed controller architectures is using a suitable test bed, e.g., Mininet[6] or DOT [20]. However, often access to existing OpenFlow test beds is limited or they do not feature the required network characteristics. Analytical approaches as presented in [10] presuppose a high abstraction level and neglect specific details, e.g., protocol behaviour on data link layer. Hence, another possible approach is to create a simulation model featuring the desired level of detail, e.g., using OMNeT++.

A detailed implementation of the OpenFlow protocol together with an investigation of the controller placement problem is provided in [12]. Although the investigated research question is closely related to the investigation of distributed controller architectures, the provided source code could not be extended due to a tight integration of the main controller and application responsibilities.

3 OpenFlow OMNeT++ Suite

To provide the tools needed to investigate the performance and impact of different SDN controller architectures in large scale networks, we implemented the OpenFlow OMNeT++ Suite, which extends the OMNeT++ framework with OpenFlow capabilities. For this, the OpenFlow OMNeT++ Suite provides an SDN controller and an OpenFlow enabled switch module, which communicate via the OpenFlow protocol. Furthermore, the suite features a variety of common controller applications such as topology discovery, different forwarding mechanisms and an ARP proxy. In addition to the common single controller approach, the package also features a horizontally and a hierarchically distributed controller architecture. The implementation of the OpenFlow OMNeT++ Suite is based upon OMNeT++ and INET in version 4.6 and 2.5.0 respectively. The OpenFlow OMNeT++ Suite's content is grouped into six packages, illustrated in Figure 1.

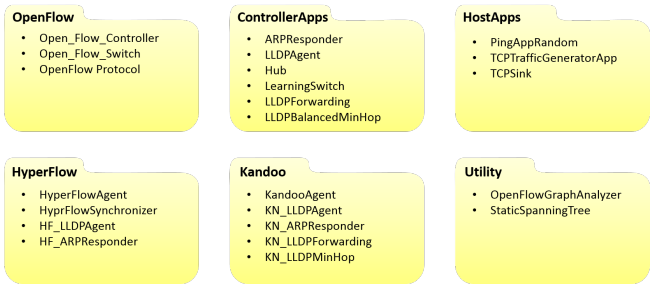


Fig. 1: OpenFlow OMNeT++ Suite Packages.

In the following, we describe the main modules in detail. The source code of the OMNeT++ Suite is publicly available [1] and is published under GPL v3 license.

3.1 OpenFlow

The OpenFlow package holds the main components, which consist of the OpenFlow protocol implementation, as well as the OpenFlow enabled switch and controller compound module.

[1] https://github.com/lsinfo3/OpenFlowOMNeTSuite

OpenFlow Protocol The communication between the SDN controller and the switch is enabled by the OpenFlow protocol, which defines a variety of messages. For the implementation of the supported OpenFlow messages, we reused the exact same message definitions as detailed by the OpenFlow Switch Specification [4] in version 1.3 as far as applicable, to model the protocol as close as possible. Since the implementation of every single OpenFlow message would be beyond the scope of this work, we focused on messages providing the main functionality of the protocol. At the current state, the suite supports *hello*, *Feature-request* and *Feature-reply* messages to initiate the OpenFlow channel as well as *Packet-in* and *Packet-out* messages to control the forwarding of data plane packets. At last, *Flow-mod* and *Port-mod* messages can be used to modify the flow or port table of an individual switch. All implemented messages are derived from the *Open_Flow_Message* class, which merely holds the OpenFlow header. Following this approach, additional message classes can be easily created if the necessity arises.

OpenFlow Switch An implementation of the OpenFlow enabled switch is represented by the *Open_Flow_Switch* compound module. As displayed in Figure 2, the module is composed from multiple components, which are organized in three groups.

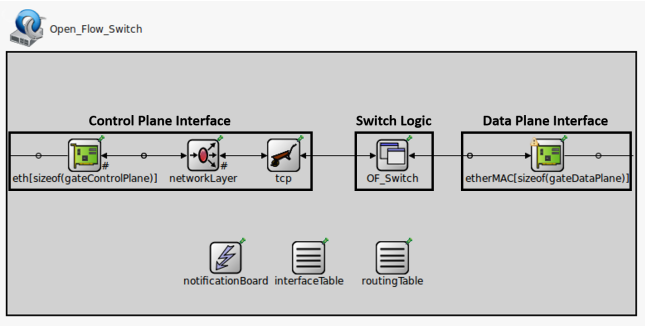


Fig. 2: Open_Flow_Switch Compound Module in OMNeT++.

Whereas the right group consists of a single gate named *gateDataPlane* and acts as the interface to the data plane, the group on the left is responsible for the communication with the control plane. Since this communication is handled via TCP, additional modules are required. The first module is a gate which functions as an interface and allows connections to and from other modules. The *networkLayer* and *tcp* modules are both provided by the INET framework and mimic the functionality of their real world counterparts. The *OF_Switch* module implements the main logic of the OpenFlow switch and thus is connected to both groups. This module is responsible for the connection initiation to the OpenFlow controller as well as managing packets received from the control and data plane.

The switch can be configured with a service time, which is used to simulate the time needed to process a packet received from the data plane. In addition, the controller's IP address and port as well as the time at which the switch should initiate the connection to the controller can be set. The OpenFlow connection is then established, by first completing the TCP 3-Way-Handshake which is followed by the initiation of the OpenFlow Channel. For this the controller and the switch first exchange *hello* messages. The controller then sends an *OFP_Feature_Request* message, to which the *OF_Switch* module responds with an *OFP_Feature_Reply*.

Once a packet originating from the data plane is scheduled for execution, the *OF_Switch* module first checks if the *flowTable* property can match the received packet. If a matching entry is returned, the packet will be handled according to the contained instructions. Otherwise, the *OF_Switch* module extracts the header fields of the received packet and builds an *OFP_PacketIn* message. It then tries to store the packet into its *buffer* property and, if successful, updates the Packet-in message with the appropriate identifier. In the case that the buffer has reached its full capacity, the entire packet is encapsulated into the Packet-in message. Finally, the Packet-in message is sent to the controller.

Whenever the *OF_Switch* receives a packet from the control plane, it casts the received packet to an *Open_Flow_Message* and then uses the OpenFlow header to determine the type of the OpenFlow message. In case of an *OFP_Packet_Out* message, the *OF_Switch* module first checks if the message contains the original frame or if the message states a buffer id. Hence, the frame is restored by either decapsulation or by retrieving it from the buffer respectively. The module then continues by applying the actions contained in the Packet-out message to the frame. If the message is a *OFP_Flow_Mod* message, the *OF_Switch* module extracts the necessary fields for constructing a new Flow entry, which is then stored by the *flowTable* property. At last *OFP_Port_Mod* messages are processed by updating the corresponding entry in the module's *portVector* property.

SDN Controller The OpenFlow OMNeT++ Suite models the functionality of the SDN controller within the *Open_Flow_Controller* compound module, which is illustrated in Figure 3. As OpenFlow is the de-facto standard southbound API protocol currently in use, we will utilize the term openFlow/SDN controller interchangeably through out this paper. The individual components can be categorized into three groups according to their functionality as depicted in Figure 3.

Similar to the *Open_Flow_Switch* module, the category on the right is composed of three modules and functions as an interface to the control plane. The *controllerApps* and *tcpControllerApps* are used as slots, which enable the user to load different applications to the controller. The individual applications can extend or alter the behaviour of the controller.The *OF_Controller* module implements the main logic of the controller and is responsible for establishing and maintaining the communication to the OpenFlow switches as well as interacting with the assigned controller applications. Following the design architecture of most real-world controllers, the *OF_Controller* module can support an arbi-

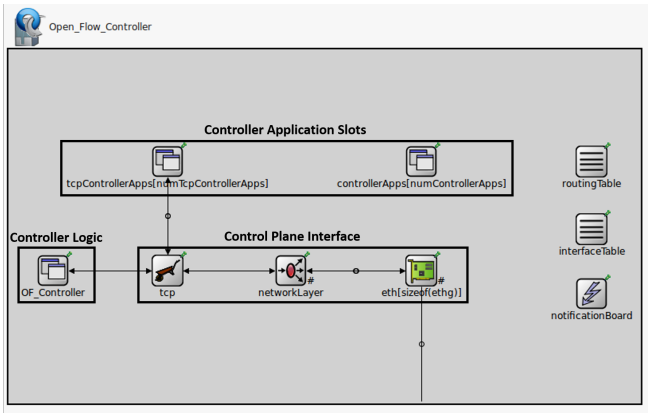


Fig. 3: Open_Flow_Controller Compund Module.

trary number of applications which run on top of the main controller process. The communication between the controller and its applications is realized by a producer/consumer design pattern provided by OMNeT++ called signals.

Prior to the start of the simulation, the *Open_Flow_Controller* can be configured with an IP address and port to listen for active OpenFlow connections. Furthermore, a service time can be set which is applied to all received packets to simulate the time needed by the controller to process a request from the switch. At last, the module emits a signal at the beginning of the simulation to inform the controller applications that it has booted and is now fully operational.

Once a packet is received from the data plane and scheduled for execution, the module checks if it has been received from an open connection or if a new connection has to be established. If the message is sent by an unestablished connection, the controller forks the initial connection and initiates the OpenFlow Channel as specified by the OpenFlow protocol. Yet, if the packet has arrived via an existing connection, the *OF_Controller* module emits a Packet-in signal with an attached reference pointer to the packet. This signal is then examined by the individual controller applications, which determine the further course of action.

Following this design pattern, a unique signal is used to inform the applications of received and sent OpenFlow messages. Each of these signals provides a pointer to the original message which triggered the signal. This message can then be examined by the application for further processing. At last, the *OF_Controller* module provides an interface to the controller applications for sending OpenFlow messages to its managed switches.

3.2 Controller Applications

The *OF_Controller* module is intentionally kept simple and only manages the connection to the switches. Hence the controller applications are required to handle more complex processes. In this context, the suite allows the user to configure

8 Nicholas Gray et al.

each individual controller to host an arbitrary number of applications, which can extend, alter or redefine the behaviour of the controller. This approach provides a high interchangeability and extensibility of existing and future modules.

ARPResponder To establish a connection to another device located on the same network, a device has to first determine the MAC address associated to the destination IP address. Typically, the device issues an ARP [19] Request, which is broadcasted to all devices on the network imposing additional load onto the network, especially in large broadcast domains.

The *ARPResponder* controller application tries to mitigate this effect, by caching IP-to-MAC address associations. By directly replying to ARP requests, the load induced by ARP flooding can be reduced.

LLDPAgent SDN controllers often rely on the Link Layer Discovery Protocol (LLDP) to build a map of the network topology [1]. To bring this functionality to OMNeT++ the OpenFlow OMNeT++ Suite provides the *LLDPAgent* controller application and a *LLDP* message class. Whereas the *LLDP* message reflects the structure of its real world counterpart, the *LLDPAgent* module sends these messages in regular time intervals.

Hub The *Hub* controller application is the simplest forwarding mechanism provided by the OpenFlow OMNeT++ Suite. It implements a hub behaviour, in which the switch is instructed to flood every data plane packet on all active ports except for the ingress port.

Learning Switch The second forwarding mechanism is implemented by the *LearningSwitch* controller application. In comparison to the *Hub* module, this controller application maps the observed MAC addresses to the respective ingress port and is thus able to directly forward data plane packets towards their destination.

LLDP Forwarding The *LLDPForwarding* controller application makes use of the topology information provided by the *LLDPAgent* module to forward packets along the shortest path from source to destination. The shortest path is computed by Dijkstra's algorithm and the hop count is used as cost function. Since the information of the network topology might be outdated or incomplete, no route to the destination may be found. In this case, the module can be either configured to drop or flood the packet. Yet, if a path is returned, the *LLDPForwarding* controller application sends a Flow-mod message to every switch along the path to establish the forwarding route. Once all Flow-mod messages have been sent, the module proceeds by sending a Packet-out message to the switch which originally triggered the request, to forward the packet to the next hop. From there on, all switches along the path have already executed the Flow-Mod instructions and are now able to directly forward the packet to its destination without any further controller intervention.

LLDP Balanced Min Hop The last forwarding mechanism featured by the OpenFlow OMNeT++ Suite is implemented by the *BalancedMinHop* module and forwards packets along the shortest path towards the destination. Whereas the *LLDPForwarding* controller application always uses the same deterministic route, the *BalancedMinHop* module tries to balance the load among several shortest paths. This not only makes better use of the provided bandwidth but also spreads the load among the intermediary network devices, thus making this forwarding mechanism especially beneficial for network topologies containing numerous redundant paths to a single node.

3.3 Host Applications

Host applications are similar to the controller applications, except that they are running on the *StandardHost* module, which is included in the INET framework. In total, the OpenFlow OMNeT++ Suite features three host applications which are presented in the following.

Ping App Random The *PingAppRandom* module extends the standard ping application provided by the INET framework. Whereas the base module sends a ping request to a preconfigured IP address, the *PingAppRandom* application selects a random host by an uniform distribution as destination.

TCP Traffic Generator The *TCPTrafficGeneratorApp* module provides the possibility to generate a realistic TCP traffic pattern. For this, the module can be configured with an arrival rate, at which it establishes a TCP connection to a random host in the network. Once the connection is established, the module starts by sending an amount of data according to a flow size value which is randomly selected from an input file.

TCP Traffic Sink The *TCPTrafficSinkApp* host application is used in combination with the *TCPTrafficGeneratorApp* module and serves as communication partner. Once the connection is established, the *TCPTrafficSinkApp* module accepts all incoming packets, but does not process them any further. In contrast to the TCPSink module provided by OMNeT++, our implementation actively closes the TCP connection on reception of the last packet. This has been done to correctly record the timestamps of the individual connection phases.

3.4 HyperFlow

In addition to a centralized controller, the OpenFlow OMNeT++ Suite provides a HyperFlow implementation, which realizes a horizontally distributed controller architecture [21]. The implementation consists of a *HyperFlowAgent* and a *Hyper_Flow_Synchronizer* compound module as well as custom messages which establish the communication between these two entities. Furthermore, selected controller applications have been expanded to denote and replay events which need to be synchronized. In the following we give details about the interior working of these modules and state the changes we made to port HyperFlow to the OMNeT++ framework.

HyperFlow Agent The *HyperFlowAgent* module is implemented as controller application and is modelled according to the original HyperFlow design [21], despite some minor differences discussed in Section 3.4. It provides an interface to the controller applications to replicate state changing events as well as synchronizing the local view in regular time intervals. Furthermore, it implements a failure discovery of controller instances.

HyperFlow Synchronizer The original HyperFlow implementation uses a distributed file system as synchronization mechanism, which is not available in OMNeT++. Thus, we created the *Hyper_Flow_Synchronizer* module, which functions as master node during the synchronization process.

In comparison to the distributed file system, the *Hyper_Flow_Synchronizer* module needs to be placed in one fixed location in the network and imposes a higher delay to nodes having a physical greater distance to this module. The distributed file system handles this issue more efficiently by distributing the information from nodes which are located in closer proximity to each other. Thus, the original HyperFlow implementation is able to distribute the information more quickly and hence can provide a smaller window of inconsistency among all nodes. Yet, the *Hyper_Flow_Synchronizer* module provides the unique ability to simulate the performance of different synchronization mechanism by setting the service time parameter to reflect the capabilities of the system of interest. Since the overall performance of HyperFlow is decisively determined by the underlying synchronization mechanism as stated in [21], the *Hyper_Flow_Synchronizer* module enables an extensive analysis of different synchronization technologies.

HyperFlow Controller Applications As HyperFlow requires a modification of the controller applications to signal state changing events, this also applies to the applications provided by the OpenFlow OMNeT++ Suite. Currently, the suite provides two controller applications, which have been adapted to work in combination with the *HyperFlowAgent* module, i.e., the *HF_ARPResponder* and *HF_LLDPAgent* module. Each of these modules are derived from their respectively named parent module and feature the same core functionality. In addition to the functionality provided by the base class, the modules have been extended to use the *HyperFlowAgent* module to denote state changing events and to synchronize their own state by listening to emitted signals from this module.

3.5 Kandoo

In addition to HyperFlow, the OpenFlow OMNeT++ Suite features a hierarchical distributed controller architecture, which is modelled after Kandoo. Here, two distinct hierarchies of controllers are utilized to balance the load inflicted onto the control plane. For this, the *KandooAgent* controller application module can be configured to either run as local or root controller. In addition, the suite provides a variety of controller applications, which have been expanded for their use with Kandoo. In the following, we outline the implementation details of these modules and describe their interactions.

Kandoo Agent The *KandooAgent* module provides the core functionality and is modelled in similarity to the original Kandoo implementation [7]. As Kandoo differentiates between local and root controllers, the module can be configured with the according mode of operation. If the application runs in local mode it provides interfaces to other controller applications to forward requests to the root controller. In case the *KandooAgent* module is set to root mode, it maintains the global view of the network and handles requests from local controllers.

Kandoo Controller Applications As stated in [7], a modification of the controller applications is required to enable the communication between the local and root controller instances. In addition, each application has to respect the mode of operation in which the controller is currently running and has to behave accordingly. In this context, we have adapted a total of four controller applications to function properly in combination with the *KandooAgent* module. Namely, these four modules are the *KN_ARPResponder*, *KN_LLDPAgent*, *KN_LLDPForwarding* and *KN_LLDPBalancedMinHop*, which are all derived from their respective base classes and feature the same core functionality.

3.6 Utility

The utility package of the OpenFlow OMNeT++ Suite contains two modules, i.e., *StaticSpanningTree* and *OpenFlowGraphAnalyzer*, which assist in constructing and analysing an OpenFlow enabled simulation network.

Static Spanning Tree The *StaticSpanningTree* module constructs a spanning tree on the data plane of the switches and thus enables simulations of network topologies containing loops. This is achieved by setting the *OFPPC_NO_FLOOD* flag on all ports, which are not part of the spanning tree.

OpenFlow Graph Analyzer The *OpenFlowGraphAnalzer* module can be placed within an OpenFlow network and extracts several graph characteristics, i.e., the minimum, maximum and the average path length.

4 Evaluation of Distributed Controller Architectures

To analyse the offered traffic of distributed controllers architectures, we modelled a topology after the Advanced Layer 2 Services (AL2S) test bed in the OpenFlow OMNeT++ Suite as illustrated in Figure 4. The AL2S test bed is comprised of multiple sites distributed throughout the United States, which are interconnected and enable researchers to investigate different configuration mechanisms by using Software-defined networking technologies such as OpenFlow.

In total the modelled topology features 34 *Open_Flow_Domain* nodes which reflect the individual sites. Each domain features one OpenFlow switch module as well as a configurable number of *StandardHost* modules, which has been set to two for our investigations. The connections between the sites have been realized

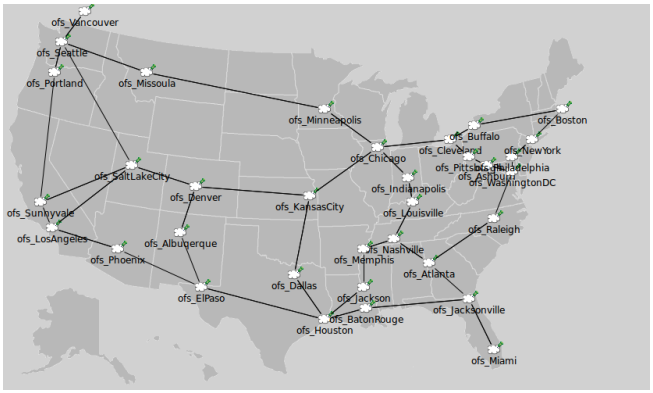


Fig. 4: Subset of the AL2S test bed in OMNeT++.

in the OMNeT++ simulation framework through the use of the *DistanceChannel*, which imposes a delay on every message according to the physical length of the channel at a data rate of $10Gbps$. We used POCO [8] to compute the controller placements based on the minimum controller to switch latency. The individual placements are detailed in Table 1.

Table 1: AL2S Controller Placement for a varying Number of k Controllers

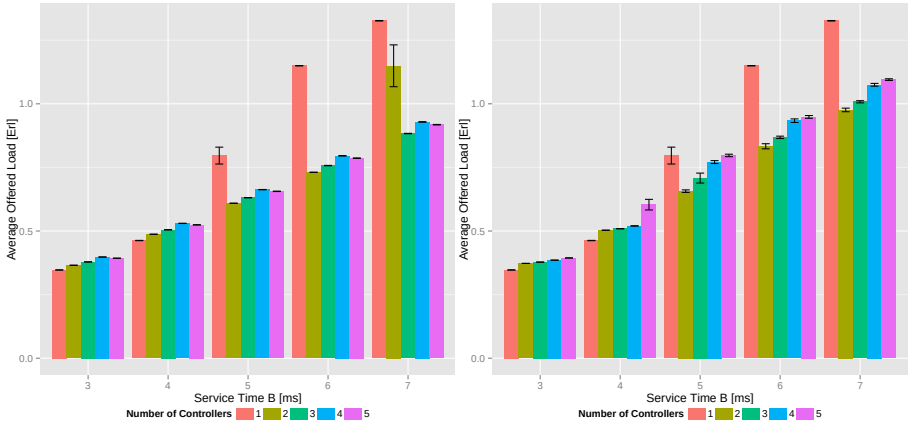
Controller/k	1	2	3	4	5
Controller 1	Kansas City	Louisville	Denver	Seattle	Seattle
Controller 2	-	Salt Lake City	Portland	El Paso	El Paso
Controller 3	-	-	Atlanta	Chicago	Atlanta
Controller 4	-	-	-	Atlanta	Houston
Controller 5	-	-	-	-	Cleveland

In our scenario, the controllers are set to run the *ARPResponder*, *LLDPAgent* and the *LLDPForwarding* module in addition to the respective distributed controller agents. The LLDPAgent module is configured to send LLDP messages every 30 seconds, which is the default value of the Cisco IOS Operating System. The StandardHost modules are all equipped with the PingAppRandom application, which is configured to send one ping request per second to a random host in the network. Furthermore, ARP responses are cached for 60 seconds to represent the default value of the Linux Mint Operating System. To conclude, the service time of each OpenFlow switch is configured to 0.035 ms, which is the average processing delay per packet as observed in [3]. For the following results we simulate a time span of 30 minutes and each configuration is repeated 4 times. In this configuration, a single simulation run is completed within approximately 12 minutes when executed on an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz having 16 GB memory.

To quantify the impact of the individual controller architectures, we investigate the offered load, which is derived by dividing the number of packets per second at the individual control plane interfaces by the controller's service time per packet.

4.1 HyperFlow

We start our evaluation of the offered load for the horizontal controller architecture by configuring all controllers with the *HyperFlowAgent* module. Here, the placement of the synchronization module resides in Kansas City. For our investigations, we vary the number of controllers from 1 to 5 as shown in Table 1 and for each configuration repeat the simulation with a deterministic message service time ranging from 3 to 7 ms. In Figure 5a, the y-axis states the offered load and the x-axis groups the varying number of controllers by the configured service time. Within one group the red, brown, green, blue and purple bar relate to the scenario in which 1, 2, 3, 4 or 5 controllers are used respectively and the error bars represent the 95% confidence interval.



(a) HyperFlow (b) Kandoo

Fig. 5: Offered Load for a varying number of controllers using (a) Hyperflow and (b) Kandoo.

For a service time of 3 and 4 ms, the offered load increases with a rising number of controllers with the exception of the 5 controller scenario. The increased load for the multi-controller scenarios is influenced by several impact factors in a non-intuitive manner. This is due to the induced synchronization load, which is augmented with each additional controller. The slight exception for 5 controllers results from a more efficient and balanced controller placement within the AL2S topology. Starting with a service time of 5 ms, the single controller scenario

14 Nicholas Gray et al.

produces the highest offered load for a given service time. This results from a single controller no longer being able to handle the traffic in a timely manner. Thus TCP timers of the host applications expire and force a retransmit, hence causing more offered load. In the case of a service time of 7 ms, this effect can also be observed for the two controller scenario.

4.2 Kandoo

We continue our investigation by analysing the impact of a hierarchical controller architecture on the offered load. For this all controllers are configured with the *KandooAgent* module and an additional root controller is placed in Kansas City. Figure 5b displays the results. As described previously, the y-axis denotes the offered load, whereas the individual service times are plotted on the x-axis. The number of configured controllers is represented by the bar color and the error bars depict the 95% confidence intervals.

Overall, the figure shows a general increase on the offered load with rising service times. For a service time of 3 ms and 4 ms, the offered load is augmented with additional controllers, which is caused by the synchronization overhead. Starting with a service time of 5 ms, the single controller scenario induces more load into the system compared to when using 2 or 3 controllers. Again, this is caused by the resource limitation of a single controller and its incapability to handle the offered traffic. Yet, a similar trend is observed for the 4 and 5 controller scenarios. This effect originates from the increased partitioning of the network when adding more controllers. As Kandoo's local controllers can only handle requests within in their own domain and have to forward all other requests to the root controller, the root controller may impose a bottleneck if it is not properly shielded by the local controllers. The efficiency of shielding the root controller is strongly related to the degree of locality contained within the traffic pattern. By adding controllers to the cluster, the size of the individual controller domains is decreased and the probability that the destination of a packet is outside of the local domain increases. Thus, increasing the number of requests to the root controller. Similar as in the single controller scenario, an overloaded root controller may stall the processing of packets to such an extend, that retransmissions are triggered and therefore the overall offered load increases. At last, for longer service times of 6 ms and 7 ms, the benefits of a multi controller implementation outweigh the partitioning overhead and hence the offered load is lower as compared to the single controller scenario.

5 Conclusion

In this paper we present the OpenFlow OMNeT++ Suite, which enables the simulation of distributed SDN controller architectures within the OMNeT++ framework. In addition to a single controller approach, the suite provides a horizontal and a hierarchical distributed controller architecture, which are modelled in resemblance to HyperFlow and Kandoo respectively. To demonstrate

the suite's capabilities, we perform an exemplary evaluation highlighting performance influence factors of these controller architectures. The results show how partitioning, instance placement and service times may affect the offered control plane load. Furthermore, a greater synchronization overhead is observed for a larger number of distributed controller instances. HyperFlow achieved a better efficiency than Kandoo in the investigated scenario. This is mostly due to the chosen traffic pattern featuring a low degree of locality. Yet, further investigations have to be conducted to determine the exact degree of locality needed by Kandoo as well as to determine further key performance indicators impacting the individual controller architectures. For these and future evaluations, the OpenFlow OMNeT++ Suite offers the necessary level of detail and flexibility to provide a solid foundation.

Acknowledgment

This work has been performed in the framework of the CELTIC EUREKA project SENDATE-PLANETS (Project ID C2015/3-1), and it is partly funded by the German BMBF (Project ID 16KIS0474). The authors alone are responsible for the content of the paper.

References

1. I. S. Association. Ieee standard for local and metropolitan area networks-station and media access control connectivity discover. `http://standards.ieee.org/getieee802/download/802.1AB-2009.pdf`. called on Sep. 01, 2015.
2. P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. Onos: Towards an open, distributed sdn os. In *Proceedings of the third Workshop on Hot Topics in Software-defined Networking*. ACM, 2014.
3. F. Dürr, T. Kohler, et al. Comparing the forwarding latency of openflow hardware and software switches. Technical report, Technical Report Computer Science 2014/04, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Parallel and Distributed Systems, Distributed Systems, 2014.
4. O. N. Foundation. Openflow switch specification. `https://www.opennetworking.org/`. called on Sep. 01, 2015.
5. I. Framework. Inet framework. `http://inet.omnetpp.org/`. called on Sep. 01, 2015.
6. N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.
7. S. Hassas Yeganeh and Y. Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first Workshop on Hot Topics in Software-defined Networks*. ACM, 2012.

8. D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia. Pareto-optimal resilient controller placement in sdn-based core networks. In *Teletraffic Congress (ITC), 2013 25th International*. IEEE, 2013.
9. S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43. ACM, 2013.
10. M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia. Modeling and performance evaluation of an openflow architecture. In *Proceedings of the 23rd international teletraffic congress*, pages 1–7. International Teletraffic Congress, 2011.
11. M. Jarschel, T. Zinner, T. Hossfeld, P. Tran-Gia, and W. Kellerer. Interfaces, attributes, and use cases: A compass for sdn. *Communications Magazine, IEEE*, 52, 2014.
12. D. Klein and M. Jarschel. An openflow extension for the omnet++ inet framework. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, pages 322–329. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013.
13. T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, 2010.
14. D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 1–6. ACM, 2012.
15. N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
16. OMNeT++. Omnet++ network simulation framework. `http://www.omnetpp.org/`. called on Sep. 01, 2015.
17. OpenDaylight. Opendaylight. `https://www.opendaylight.org/`. called on Sep. 01, 2015.
18. K. Phemius, M. Bouet, and J. Leguay. Disco: Distributed multi-domain sdn controllers. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–4. IEEE, 2014.
19. D. Plummer. Ethernet address resolution protocol: Or converting network protocol addresses to 48. bit ethernet address for transmission on ethernet hardware. *Request For Comments 826*, 1982.
20. A. R. Roy, M. F. Bari, M. F. Zhani, R. Ahmed, and R. Boutaba. Design and Management of DOT: A Distributed OpenFlow Testbed. In *14th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, May 2014.
21. A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*. USENIX Association, 2010.
22. V. Yazici, M. O. Sunay, and A. O. Ercan. Controlling a software-defined network via distributed controllers. *arXiv preprint arXiv:1401.7651*, 2014.