Journal of Network and Computer Applications 156 (2020) 102563

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Review

A comprehensive survey of interface protocols for software defined networks

Zohaib Latif[a], Kashif Sharif[a,b,*], Fan Li[a,b,**], Md Monjurul Karim[a], Sujit Biswas[a], Yu Wang[c]

[a] School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China
[b] Beijing Engineering Research Center of Massive Language Information Processing and Cloud Computing Application, Beijing, China
[c] Department of Computer and Information Sciences, Temple University, Philadelphia, USA

ARTICLE INFO

Keywords:
Software defined networks
SDN interfaces
Southbound interface
Northbound interface
East/westbound interface

ABSTRACT

Software Defined Network implementation has seen tremendous growth and deployment in different types of networks. Compared to traditional networks it decouples the control logic from network layer devices and centralizes it for efficient traffic forwarding and flow management across the domain. This multi-layered architecture has data forwarding devices at the bottom in the data plane, which is programmed by controllers in the control plane. The high-level management plane interacts with the control plane to program the whole network and enforce different policies. The interaction among these planes is done through interfaces that work as communication/programming protocols. In this survey, we present a comprehensive study of these interface and programming protocols, which are primarily classified into southbound, northbound, and east/westbound interfaces. This work first classifies each of them into subcategories and then presents a comprehensive comparative analysis. As the different interfaces have different properties, hence, the sub-classification and their analysis are done using different properties. In addition, we also discuss the impact of different virtualization techniques, such as hypervisors, on interface protocols and inter-plane communication. More over specialized interfaces for emerging technologies such as the Internet of Things and wireless sensor networks are also presented. Finally, the paper highlights several short term and long term research challenges and open issues specific to the SDN interface protocols.

1. Introduction

Over 4 billion Internet users are connected through almost 65,000 Autonomous Systems (AS) throughout the world, and increasing rapidly every year (Kemp, 2018; Yangyang and Jun 2020). Every AS requires a set of applications to manage these networks. The implementation of this diverse range of applications becomes difficult by using traditional network elements. These elements are usually based on Application Specific Integrated Circuits, which may be vendor specific and requires embedded OS with hundreds of lines of code in low-level languages. Configuration and implementation of policies on these devices is not only time consuming but also difficult. Furthermore, it introduces rigidity in networks, due to the application-specific nature of devices, which makes network management complex.

Software Defined Networks (SDN) (Kreutz et al., 2015) is an emerging form of networks that promises to resolve these issues by decoupling the control plane from the data plane and provides a software-based centralized controller. By this separation of control plane and data plane, network switches become simple forwarding devices. Whereas, decision making is shifted to the controller, which provides a global view of the network and programming abstractions. This centralized entity provides programmatic and real-time control of underlying networks and devices to operators. By using SDN, network management becomes straightforward and helps in removing rigidity from the network.

The layered structure of SDN architecture, as shown in Fig. 1 has three major planes as data plane, control plane, and management plane. Data plane contains physical network elements, which form the data path. The control plane has a Network Operating System

[*] Corresponding author. School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China.
[**] Corresponding author. School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China.
E-mail address: kashif@bit.edu.cn (K. Sharif).

https://doi.org/10.1016/j.jnca.2020.102563
Received 4 July 2019; Received in revised form 17 January 2020; Accepted 2 February 2020
Available online 7 February 2020

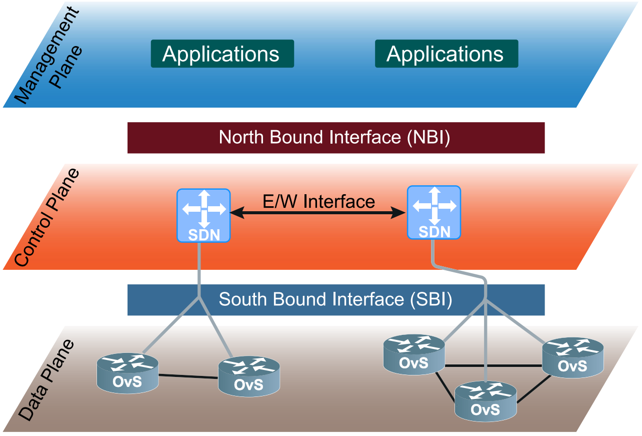


Fig. 1. Layered view of SDN Architecture and the interfaces.

(NOS), also referred to as a controller, which implements rules on data plane devices. These rules and policies are designed in the management plane of SDN architecture. Communication among these planes is established by using well-defined Application Programmable Interfaces (APIs). These interfaces are divided into Southbound, Northbound, East, and Westbound APIs. The control plane and data plane communicates through the Southbound API, which enables flow installation and configuration of devices. The control plane and management plane uses Northbound API to provide programmability in SDN. Inter-controller communication of SDN domains is established using Eastbound API, whereas Westbound API is responsible for the legacy domain to SDN domain communication.

This paper specifically deals with interface protocols for SDN. These interfaces play an important role in the overall communication process between different planes and components. The southbound interface can be segregated into OpenFlow (OF) (McKeown et al., 2008), OF dependent, and OF independent proposals. OF is the most commonly used southbound interface (SBI) in research and commercial SDNs. Extensions of SBI into emerging technologies, such as sensor networks and the Internet of Things, can also be treated as a special class of SBIs, due to their unique requirements. More precisely, IoT & sensor devices act as a perception plane, which can be visualized as a non-OF compliant extension of data plane (Ojo et al., 2016). The devices are usually connected to data plane through gateways. Northbound interfaces are classified in terms of portability, programmability, controller based, and intent-based solutions. In this article, we consider virtualization as middleware between different layers and the hardware. Hence, the interaction of APIs with different virtualization techniques requires separate classification. Eastbound interfaces are categorized in distributed and hierarchical architectures due to placement and communication of controllers, whereas westbound interfaces usually use traditional Border Gateway protocols to bridge the gap between SDN and traditional networks.

1.1. Objective & contributions of this work

Different aspects of software defined networks have been surveyed in the past, but to the best of our knowledge, there is no comparative analysis or survey which is specific to the different interface protocols. There are a significant number of interface protocols proposed in the literature and used in the industry. This work comprehensively covers all such protocols, while presenting a detailed comparative analysis. The significant contributions of this work are as follows.

- The original contribution of this article is to present a systematic survey and classification of different interface protocols for software-defined networks.
- The most commonly used southbound interface is OpenFlow, and this work classifies various solutions for southbound communication as dependent and independent of OpenFlow while comparing them for their properties and advantages.
- We also present OpenFlow like solutions for perception plane, i.e. wireless sensor networks & the Internet of Things. It is important to note that IoT (and sensor) networks are below the control plane, hence they only interact with SBIs.
- We classify the northbound interfaces based on portability, programmability, controller-based, and intent-based solutions, and individually analyze the available solutions.
- SDN combined with virtualization is a new paradigm and we analyze both southbound and northbound interfaces with respect to the virtualization platforms.
- We also categorize & discuss east and westbound APIs for communication between SDN domains and legacy networks.
- Finally, we discuss future research directions for each of the interfaces separately in the end.

1.2. Existing SDN studies

A comprehensive study of SDN is a difficult task as it is a multidimensional field. However, it has been explored thoroughly from different aspects in several studies. Works in (Masoudi and Ghaffari, 2016; Gong et al., 2015; Cox et al., 2017; Karakus and Durresi, 2017; Hu et al., 2018) have discussed SDN for its application, architecture, control and application plane, component design, evaluation testbeds, simulation environments, and challenges. Works in (Karakus and Durresi, 2017; Hu et al., 2018; Fonseca and Mota, 2017; Trois et al., 2016) have studied control plane scalability, consistency, reliability, controller distribution, load balancing, fault tolerance, and programming languages. In (Lu et al., 2019), authors classify controller placement problem into four aspects which include; reliability, latency, cost, and multi-objectiveness. Moreover, it provides an analysis of specific algorithms in different network scenarios. Research regarding emerging technologies in SDN, such as WSNs and IoT are discussed in (Kobo et al., 2017; Ali et al., 2017; Bera et al., 2017). In (Blenk et al., 2016), a comprehensive survey on hypervisors is presented, where different hypervisors are categorized according to their architectures and execution platforms for SDN, while in (Li and Chen, 2015), authors present a relationship between NFV and SDN and highlight the main challenges in Software Defined NFV architecture. However, the focus of this work is on the interface protocols and APIs in the layered structure and their associated challenges.

1.3. Organization of paper

The paper is organized into eight sections, as shown in Fig. 2, where section 2 describes background information related to SDN interfaces and their interaction with emerging technologies and virtualization techniques. OpenFlow and other proposals for Southbound Interfaces are presented in Section 3. Northbound interfaces and their classification is presented in Section 4 with the aspects of portability and programmability. The effects of virtualization on SDN interfaces are discussed in Section 5. Communication among SDN domains and with legacy networks is presented in Section 6. Section 7 outlines future directions, and conclusions are drawn in 8.

2. Background

The main objective of this paper is to study the interfaces among different planes of SDN. Before delving into such details, we first give an overview of SDN architecture, its different components, and their functionalities.

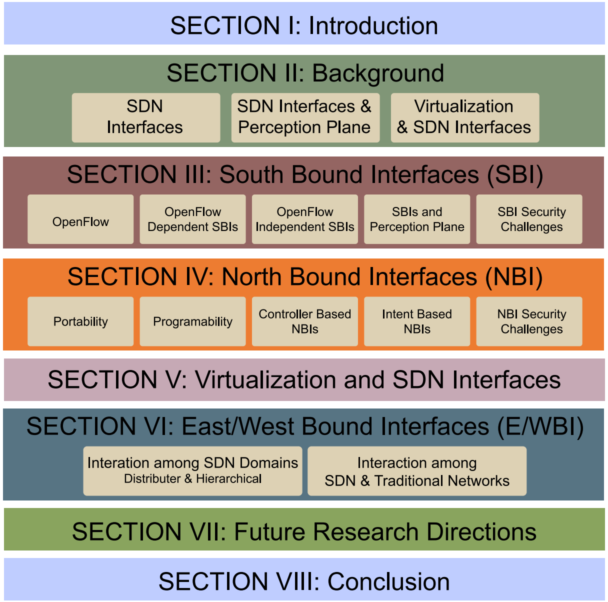

SECTION I: Introduction
SECTION II: Background
SDN Interfaces
SDN Interfaces & Perception Plane
Virtualization & SDN Interfaces
SECTION III: South Bound Interfaces (SBI)
OpenFlow
OpenFlow Dependent SBIs
OpenFlow Independent SBIs
SBIs and Perception Plane
SBI Security Challenges
SECTION IV: North Bound Interfaces (NBI)
Portability
Programability
Controller Based NBIs
Intent Based NBIs
NBI Security Challenges
SECTION V: Virtualization and SDN Interfaces
SECTION VI: East/West Bound Interfaces (E/WBI)
Interation among SDN Domains Distributer & Hierarchical
Interaction among SDN & Traditional Networks
SECTION VII: Future Research Directions
SECTION VIII: Conclusion

Fig. 2. Overview and structure of this survey.

In traditional networking, control functionalities and data forwarding elements are embedded in a single device. Due to tightly coupled control and data planes, network control is distributed in these devices, which makes it difficult to run applications on these elements due to vendor and language specifications. Moreover, the management and configuration of such devices become difficult with increasing scalability. SDN paradigm breaks this tight coupling of control functions and forwarding elements and provides a unified control for running different types of applications (Kreutz et al., 2015). presents comprehensive details on the working and architecture of SDNs, hence, we focus only on the interface details.

Before SDN, the concept of network programmability was studied from two aspects: Active Networking (Tennenhouse et al., 1997) and Programmable Networks (Campbell et al., 1999) (A&PN). Active Networking discusses the injection of intelligence in the network beyond the conventional processing of packets. Whereas, Programmable Networks allows the control of network device behavior and flow control through software. This lays a clear foundation for the separation between data and control plane, which later combined to become the Software Defined Networking paradigm. The main reason for being a wide acceptance of this paradigm is the rapid innovation in both planes (i.e. control plane & data plane) (Haleplidis et al., 2015).

A controller is the fundamental component of the SDN control plane. One of the key roles of a controller is to manage the traffic in underlying network elements by using a set of instructions. There is a wide range of controllers in SDN, and work in (Zhu et al., 2019) presents a comprehensive review of their capabilities. Features of each controller may differ from one another, but the core and essential functionalities of all the controllers are similar, for example, topology information, statistics, notifications, and device management. To perform these tasks, every controller uses a southbound interface such as OpenFlow. However, some of the controllers offer a wide range of southbound interfaces (e.g. OpenDaylight). Similarly, to run various applications, every controller offers a northbound interface, however, there is no standardized interface for this communication. Some data plane architectures have multi-controller capabilities, which require eastbound communication interfaces among the distributed controllers, whereas controllers may try to communicate with legacy routers using the westbound interface.

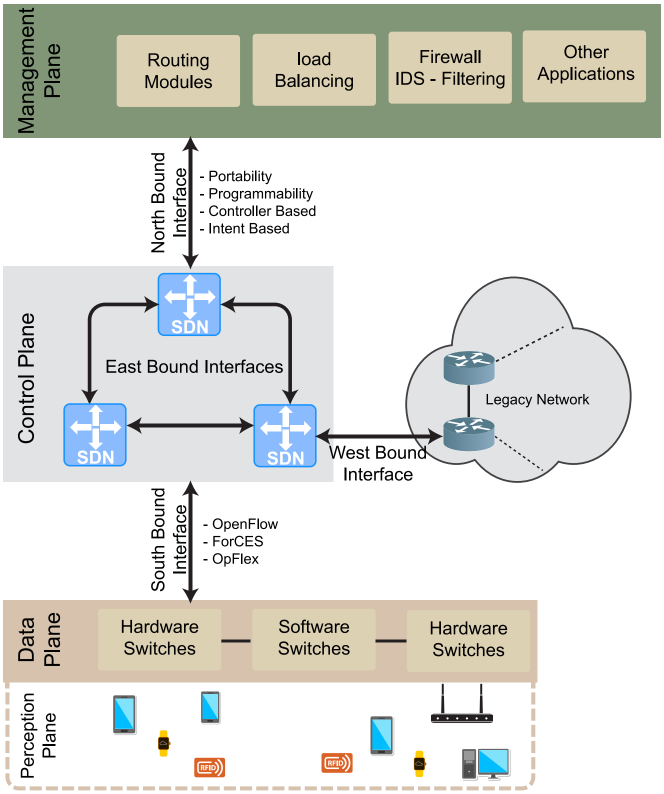


Fig. 3. SDN interface placement and properties.

2.1. SDN interfaces

APIs are an architectural component of SDN used to push configurations or information to forwarding elements or applications respectively. Fig. 3 shows the different interface APIs and some of their properties.

2.1.1. Southbound API

Southbound API (SBI) is an SDN enabler, which provides a communication protocol between the control plane and the data plane. This API is used to push configuration information and install flow entries in the data plane. It also provides an abstraction of the network device's functionality to the control plane. Major challenges of southbound interfaces are heterogeneity, vendor-specific network elements, and language specifications. The southbound interface in SDN resolves these issues by providing an open and standardized interface. There are several examples for Southbound APIs, however OpenFlow (McKeown et al., 2008) is considered as a standard in SDN. Section 3 presents a detailed review of different SBIs.

2.1.2. Northbound API

The numerous benefits of SDN are fruitless if applications can not benefit. SDN adoption depends on its ability to support a wide range of applications. Northbound APIs (NBIs) play an integral role for application developers and provides a common interface between controller and management plane, by providing the information of underlying devices for application development. Unlike the southbound interface, the northbound interface has seen fewer standardization efforts (Kreutz et al., 2015). A wide range of NBIs is offered by current controllers and programming languages. In this paper, we discuss them from different aspects in Sections 4 and 5.

2.1.3. East & westbound API

Centralized control over the network is the key feature of SDN, but only a limited number of switches can be handled by a single controller. In large scale networks, distributed controllers become a requirement, where every controller has its domain with underlying forwarding devices. Controllers need to share information about their respective domains for a consistent global view of the entire network. Eastbound APIs are used to import and export information among distributed controllers. Some examples of these interfaces are (Yin et al., 2012; AMQP, 2020; RabbitMQ, 2020). On the other hand, Westbound APIs enable the communication between legacy network devices (routers) with the controllers. Some example solutions are discussed in (Nascimento et al., 2011; Lin et al., 2013, 2014a). Detailed discussion and review are given in Section 6.

2.2. SDN interfaces and perception plane

The use of SDN in domains other than local networks and data centers can be beneficial as well as challenging.

Wireless Sensor Networks (WSNs) have been deployed in widespread applications from civil to military, and from environmental to health care. In more recent years, the Internet of Things (IoT) (Siow et al., 2018) has evolved as a major future Internet component (Bradley, 2013), where WSNs are integrated into a larger ecosystem with heterogeneous devices and use cases. SDN can provide centralized control and configuration, policy enforcement, and programming abstraction, for large scale IoT (and sensor) networks (Boulis et al., 2007; Mottola and Picco, 2011). Although some research has been done for the softwarization of WSNs (Kobo et al., 2017; Khan et al., 2016), and IoT (Bizanis and Kuipers, 2016), however, it is limited to architecture and security perspective. Most of IoT and sensor networks are visualized as a perception layer below data plane in the SDN architecture, as shown in Fig. 3. A major challenge is to extend the SBI into the perception plane beyond OF switches.

2.3. Virtualization and SDN interfaces

Network Function Virtualization (NFV) (Mijumbi et al., 2016) offers new ways to deploy, design, and manage networking devices. It separates network functionality, such as firewalls, Network Address Translation (NAT), and Domain Name Service (DNS), from hardware devices, so that they can be run remotely. There are several studies (Wang et al., 2016a; Fei et al., 2017, 2018; Zeng et al., 2018; Wang et al., 2017a; Li et al., 2018; Xu et al., 2016) on NFV placement, scheduling, routing, load balancing, 5G applications, performance interference, acceleration via FPGA, and energy efficiency. Similarly by using hypervisors, devices, hardware resources, and entire network slices can also be virtualized. Fig. 4 shows the interaction of NFV elements within the SDN architecture and communication with different planes. It is important to note that, SDN, NFV, and other softwarization techniques are not dependent on each other, but are rather complementary. Virtual Infrastructure Manager (VIM) controls and manages NFV infrastructure storage, computing, and network resources. It also keeps a mapping of the allocation of virtual resources to physical resources and manages virtual networks, links, and ports. Virtual Network Function Manager (VNFM) is capable of handling multiple Virtual Network Functions (VNFs) by using the Element Management System (EMS). NFV can be utilized in two ways. One is the virtualization of network resources by making slices (e.g. Hypervisors) where the southbound interface is involved. The other is to control these slices (e.g. Applications based Virtualization) which involves the northbound interface. Both of these interfaces (Northbound and Southbound) then become an integral part of NFV, while still being utilized by the SDN controller for flow installation and application communication.

3. Southbound interfaces (SBI) in SDN

Software Defined Networking separates network control and forwarding functions. With the help of southbound interfaces, forwarding function is kept on the device whereas, network control is shifted to an external controller. Southbound APIs enable the link between the data plane and the control plane. It is very important that this link remains available and secure, otherwise, the forwarding elements cannot function.

The main objective of this interface is to push notifications given by the controller, to data plane devices, and provide information about these devices to the controller. It allows the discovery of network topology, define network flows, and implement requests sent by the management plane. Some of commonly known southbound interfaces are OpenFlow (OF) (McKeown et al., 2008), FORwarding & Control Element Separation (ForCES) (Haleplidis et al., 2015), Open virtual Switch Database (OvSDB) (Pfaff and Davie, 2013), Protocol Oblivious Forwarding (POF) (Song, 2013), OpFlex (Smith et al., 2020), OpenState (Bianchi et al., 2014).

Fig. 5 presents the classification of different southbound interface proposals. OpenFlow (McKeown et al., 2008) is the most commonly used API and considered as a standard southbound interface. Most of the proposals are either extensions or somehow dependent upon OpenFlow. Two proposals for southbound interfaces (i.e. OvSDB and OF-Config) work as OpenFlow companions and help it to provide configuration capabilities. Whereas, proposals like ForCES, OpFlex, and NetConf, are independent of OpenFlow. Proposals like Sensor OpenFlow (SOF) (Luo et al., 2012), Software Defined Wireless Networks (SDWN) (Costanzo et al., 2012), SDN for WIreless SEnsors (SDN-WISE) (Galluccio et al., 2015), and HUBsFlow (Cicioğlu and ÇalhanHubsflow, 2019) are southbound interfaces defined specifically for perception plane. In the following subsections, the same classification is discussed, with an additional section on the security perspective of SBI.

3.1. OpenFlow

OpenFlow is a standardized and most commonly used southbound interface. It was designed particularly for SDN to provide communication between controller and forwarding elements. OpenFlow has evolved from version 1.0 with only 12 fixed matching fields and a single flow table to version 1.5 with 41 matching fields and several new functionalities. Fig. 6 shows the difference & capabilities of different OpenFlow versions, while work in (Ching-Hao and Lin, 2015) presents technical details of each capability.

OpenFlow enabled devices must have three main components; Flow Table, Secure Channel, and OpenFlow protocol. Devices may have one or more flow tables, while the secure channel connects them with the controller, and the protocol provides communication with external controllers as shown in Fig. 7. OpenFlow pipeline has several flow tables along with the group table and meter table. Tables in OpenFlow enabled switches have flow entries in the format of match, actions, and statistics. For each packet, header matching is done which includes; source and destination IP, source and destination port, source and destination MAC, along with VLAN tags, and Ethernet types. Flow tables are normally numbered and start from 0 and the packet processing pipeline always starts from this table. Based on this matching a particular action is taken to forward the packet on one or more ports. If no match is found, then it is forwarded to the controller using *Packet_IN* message. This message contains the information of the ingress port, packet header, and *Buffer_ID* where the packet is stored. To respond *Packet_IN* message, controller sends a *Packet_OUT* message. This message contains *Buffer_ID* of corresponding *Packet_IN* message and actions to perform (e.g. Forward to a particular port, drop, etc.). To handle the subsequent packets of the same flow, the controller sends a *Flow_Mod* message to switch with the instruction to insert rules into the flow table. Given

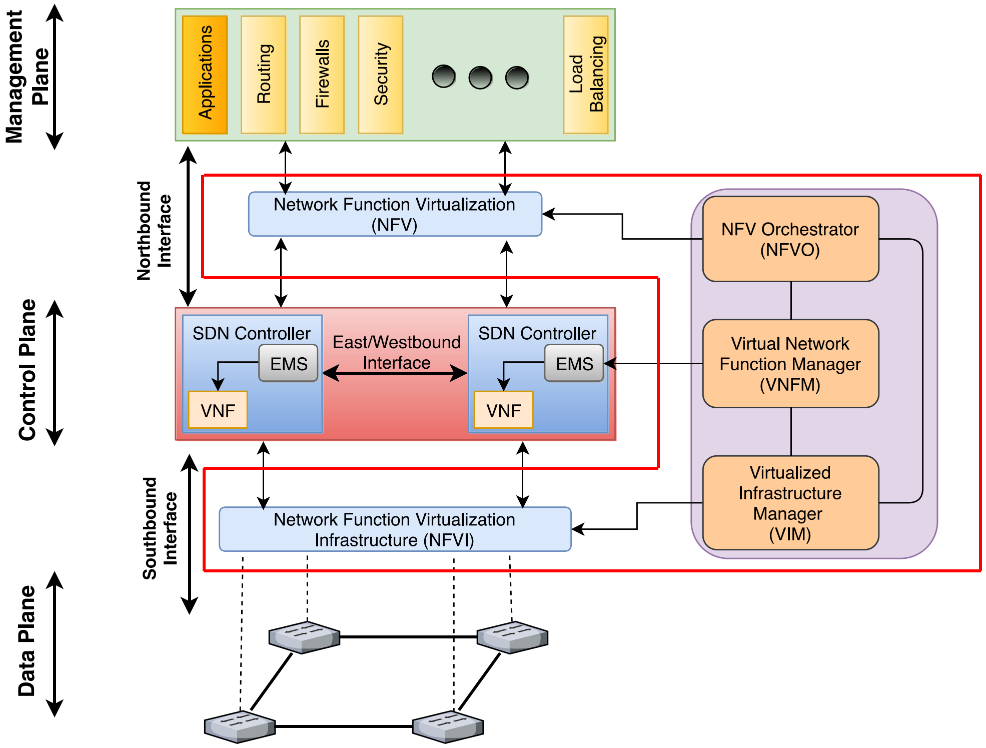


Fig. 4. Basic SDN and NFV interface abstraction.

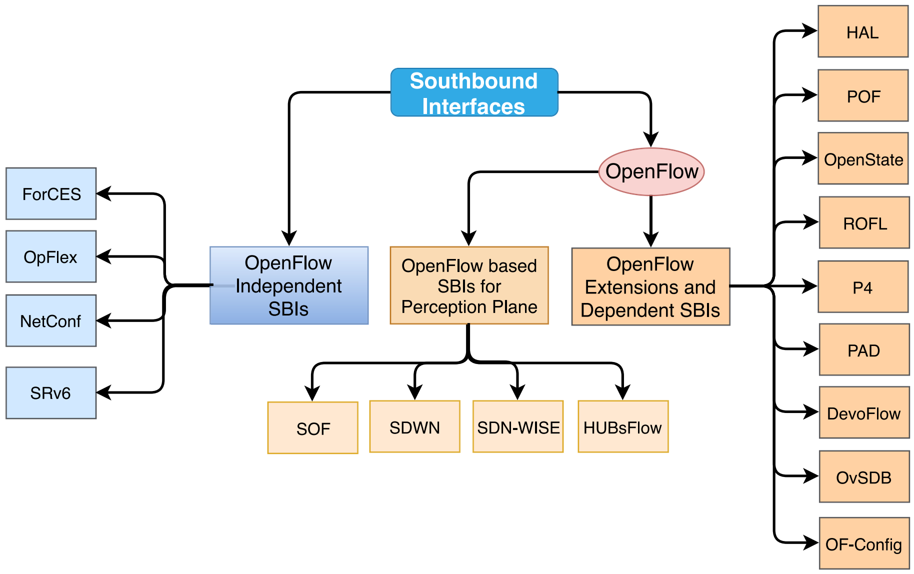


Fig. 5. Classification of OpenFlow dependent and independent SBI proposals.

rules are matched for the subsequent packets of the same flow and action is taken at line speed. Meanwhile, the counter is updated accordingly and statistics are generated per rule, per table, per port, per queue or per timer.

3.2. OpenFlow dependent SBI proposals

This sub-section describes the Southbound proposals which are based on OpenFlow and attempt to enhance its existing features or in its newer versions. Some of these issues have been addressed by OpenFlow

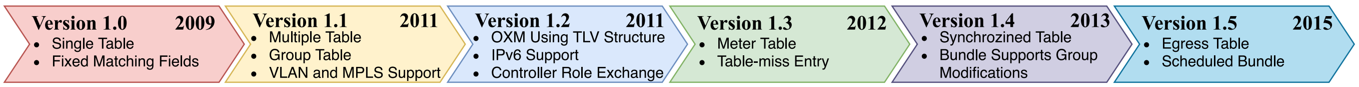
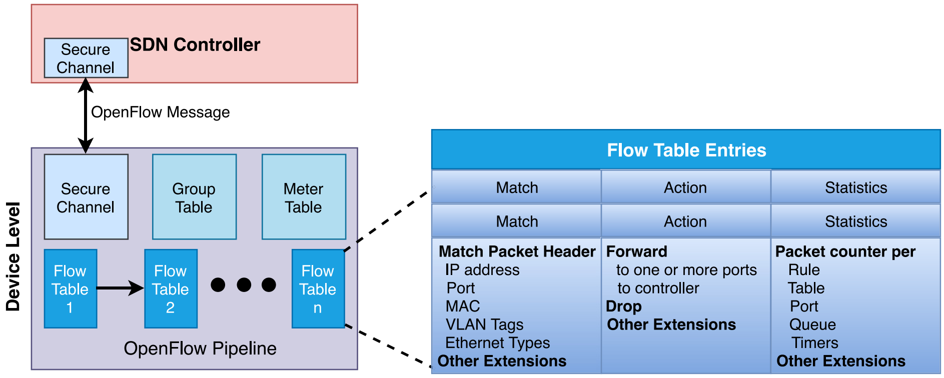


Fig. 6. Major properties of different OpenFlow versions.



Flow Table Entries		
Match	Action	Statistics
Match	Action	Statistics
Match Packet Header IP address Port MAC VLAN Tags Ethernet Types **Other Extensions**	**Forward** to one or more ports to controller **Drop** **Other Extensions**	**Packet counter per** Rule Table Port Queue Timers **Other Extensions**

Fig. 7. OpenFlow structure.

fully, some are resolved partially, and few of the shortcomings are still present in OpenFlow protocol. Table 1 represents such solutions along with their objectives.

Enabling legacy network devices to become OpenFlow compliant is a complicated task. Hardware Abstraction Layer (HAL) (Parniewicz et al., 2014) attempts to resolve this issue. It decouples hardware-specific control and management logic from the network node abstraction, which hides device complexity and vendor specific features. HAL achieved this decoupling by introducing two sub-layers: Cross Hardware Platform Layer (CHPL) and Hardware Specific Layer (HSL). CHPL covers node abstraction, virtualization, and configuration mechanisms, whereas HSL is responsible for discovering the particular hardware platform and perform all required configuration using Hardware Specific Module (HSM). Furthermore, every network element in this environment has its protocol for communication purposes, controls, and management of the underlying system. HSL is used to hide this complexity and heterogeneity. Depending on the type of network devices, communication between these two layers is done by using the Abstract Forwarding API and Hardware Pipeline API. Another interesting feature of HAL is its compatibility with multiple versions of OpenFlow.

OpenFlow has undergone many changes since its initial versions and pace of development has required other third-party hardware and software in data and control planes to make massive changes to their solutions. OpenFlow has detailed specification documents for each version but to create new libraries for each platform is time-consuming. Revised OpenFlow Library (ROFL) (Suñé et al., 2014) resolved this problem and provided a clean and easy to use API which hides the details of respective protocol versions (i.e. 1.0, 1.2 and 1.3), and simplifies application development. ROFL uses eXtensible Datapath daemon (xDPd) which is a framework for developing SDN datapath elements. It uses three major libraries, ROFL-common, ROFL-pipeline, and ROFL-HAL. ROFL-common is used to provide the basic support of OpenFlow protocol which comprises of protocol parsers and message mangling. ROFL-pipeline is employed as a data model whereas ROFL-HAL is implemented as an interface.

DevoFlow (Curtis et al., 2011) addressed the overhead created by the OpenFlow, due to full control and visibility of all flows through the software controller. It claims that the ratio of the control plane to the data plane is four orders of magnitude and less than its aggregate forwarding rate. DevoFlow attempts at resolving this problem by devolving control of most flows back to switches while the controller maintains control over targeted and significant flows only. In this way, switch-controller interactions and Ternary Content Addressable Memory (TCAM) entries may reduce overhead. Another target is to provide the aggregated flow statistics to maintain enough visibility of network, but this approach required major modifications in switch design which may be costly.

The same problem was addressed by OpenState (Bianchi et al., 2014), where authors argue that all control should not be given to a centralized controller and making switch stateless is a compromise rather than a choice which causes extra communication between devices and controller. It also suggests that the programmers can deploy states in the device rather than using an external controller. OpenState abstraction relies on the Extended Finite State Machine (XFSM) that allows the implementation of several stateful tasks inside forwarding devices. It uses XFSM as an extension of the OpenFlow match-action phenomenon and allows the implementation of several stateful tasks inside the forwarding element without introducing overhead for the controller. All the tasks, which involve local states, such as port knocking and MAC learning, can be executed directly in network elements without any overhead of control plane communication or processing delay.

Another problem in OpenFlow reported by POF (Song, 2013) is that it is reactive rather than proactive, and the data plane needs to be protocol aware. Due to this reason, data plane devices need to understand packet headers in a specific format to extract keys and execute packet processing, which again causes overhead. Moreover, the data plane is almost stateless and cannot perform any action without the involvement of controller, which means data and control planes are not properly decoupled hence leading to problems like a hindrance in innovation, reducing programmability potential, and causing complexity for large scale networks. To resolve all these problems POF proposed Flow Instruction Set (FIS) which makes forwarding elements as white boxes, protocol oblivious and ensures its simplicity. FIS is a protocol independent set of instructions that helps to compose network services from the control plane. At the same time, it helps in completely decoupling control and data plane so that both of these planes can evolve independently.

P4 (Bosshart et al., 2014) is another proposal that works in conjunction with OpenFlow. It argues that rather than extending OpenFlow specifications repeatedly, it is better to have a flexible mechanism to parse packets and match header fields. It acts as a general interface between the control plane and data plane and raises the level of abstraction for programming. P4 supports a programmable parser where new headers can be defined. Match & action stages in OpenFlow are in series whereas, P4 supports these stages in series as well as parallel. Moreover, similar to POF, it solves the issue of protocol dependency. Several studies have been done based on P4 which are integrated into other technologies and not directly related to SBIs. For example, Tango (Lazaris et al., 2014) is a control system to optimize the SDN controller for switch diversity. Similarly, Dasu et al. (2017) proposed the addition of geo-tags to IP packets to provide location-based services and enhance the capabilities of communication.

In Programming Abstraction Datapath (PAD) (Belter et al., 2014) authors discuss switch capabilities for programmability and provide a southbound API for other types of devices such as Optical Switches. It provides generic programming of forwarding devices by using byte operations that define protocol headers and functions. A packet received at the ingress port using PAD is bound with metadata and pro-

Table 1
Comparison of OpenFlow dependent SBI proposals.

Literature	Objective	Solution	Benefits	Network Type	Resolved by OpenFlow?
POF (Song, 2013)	Remove dependency on protocol specific configuration	Flow Instruction Set (FIS)	Reduce network cost by using commodity forwarding elements	Not Mentioned	No
OpenState (Bianchi et al., 2014)	Reduce Overhead between the switch and controller	Making devices as eXtensible Finite State Machines (XFSM)	Reduce Interaction between controller and forwarding elements	Not Mentioned	Partially Solved
ROFL (Suñé et al., 2014)	Development of enabled applications and support new OF versions and extensions	eXtensible DataPath daemon (XDPd)	Supports multiple versions of simultaneously	Not Mentioned	No
HAL (Parniewicz et al., 2014)	To realize OF functionality in legacy network devices	Cross Hardware Platform Layer and Hardware Specific Layer	Legacy Network devices to communicate with SDN domains	Data Center Networks	Yes
PAD (Belter et al., 2014)	Expose switch capabilities and better utilization of network devices	Generic Byte Operation	Works for devices which are not working on packets e.g. Optical Switches	Not Mentioned	Yes
DevoFlow (Curtis et al., 2011)	Control traffic overhead	Giving control to switch and controller only manages targeted flows	Reduce Interaction between controller and forwarding elements but major changes required in switch	Data Center Networks	Partially Solved
P4 (Bosshart et al., 2014)	Switch diversity and addition of new tags	Programmable packet parser	Programmers can add new tags and no specifications of OpenFlow required	Data Center Networks	No
OvSDB (Pfaff and Davie, 2020) *(For Config.)*	Enhances configuration capabilities of OpenFlow	Uses database server and switch daemon	Provides better configuration	Hybrid Networks	–
OF-Config (ONF-TS-016, 2014) *(For Config.)*	Remote configuration of OpenFlow switches	Uses configuration points	Provides flexibility for configuration in OpenFlow switches	Hybrid Networks	–

cessed through a search engine, which is a functional component of PAD. As a result of this search, a function name is added which will be executed on this packet in an execution engine. Finally, the packet is forwarded to an egress port for transmission. PAD is applicable to optical flows, where forwarding functions do not contain packet information but other instructions.

Configuration: There are two solutions that provide a configuration in OpenFlow: OvSDB and OF-Config. These protocols form a relationship between controller and switches. OpenFlow determines the route of the packet but it does not provide the management and configuration which is necessary to assign IP addresses or port allocation. In traditional networks, vendors normally use different configurations and management methods which either depend on protocols like SNMP or use the command-line interface. SDN provides a holistic view of every component of the network to engineers.

OvSDB (Pfaff and Davie, 2020) is designed to be used as a virtual switch to forward traffic between different virtual environments. It is an open-source switch, hence open to programmatic extensions and control using OpenFlow, and based on client and server implementation. Open vSwitch is a complementary protocol to OpenFlow. Inside a virtual switch, there is ovsdb-server, ovsdb-daemon and optionally a forwarding path. A virtual switch uses OpenFlow as an interface to communicate with the control and management cluster. Furthermore, it allows the creation of multiple virtual switch instances, set Quality of Service (QoS) policies on interfaces, and collect stats. Managers can specify the number of virtual bridges by using OvSDB which allows them to create, configure, and delete ports.

OpenFlow Configuration Protocol (OF-Config) (ONF-TS-016, 2014) has a special set of rules to define the mechanism for controllers to access and modify the configuration data on OpenFlow switches. It works as a companion of OpenFlow protocol and allows remote configuration of OpenFlow switches. The major difference between OpenFlow and OF-Config is that OF modifies match-action rules which effects flow in switch datapath. Whereas, OF-Config remotely configures multiple OF datapaths in a physical and virtual platform.

Insights: Most of the solutions based on (or extensions of) OpenFlow address the shortcomings in it. Some of these solutions have already been adopted by OpenFlow, whereas some of these are still open challenges. The key insight is that two main factors dominate OF research: Legacy device compatibility, and control overhead. OF developed generic solutions similar to HAL (Parniewicz et al., 2014) & PAD (Belter et al., 2014) for compatibility, and DevoFlow (Curtis et al., 2011) for control overhead reduction, which was for specific types of networks as shown in Table 3. OpenFlow version 1.3 (ONF-TS-006, 2012) addressed compatibility by introducing OpenFlow hybrid, while version 1.4 (ONF-TS-012, 2013) has added support to optical switches. To reduce the overhead between controller and data plane devices OpenFlow proposed Stats-Trigger in version 1.5 (ONF-TS-025, 2015) which solved the problem partially. However, issues like the support of multiple versions of OpenFlow and protocol dependency are still open research challenges.

3.3. OpenFlow independent SBI proposals

In this sub-section, southbound API proposals that are independent of OpenFlow or are parallel proposals have been discussed. Table 3 presents a summary of all these solutions with their objectives, solutions, and benefits.

Forwarding and Control Element Separation (ForCES) (Haleplidis et al., 2015) standardized by IETF, is a proposal which is designed to replace OpenFlow. It defines two entities as Control Element (CE) and Forwarding Element (FE), which are logically kept in the same physical device without changing the architecture of traditional networks and without the involvement of an external controller as shown in Fig. 8.

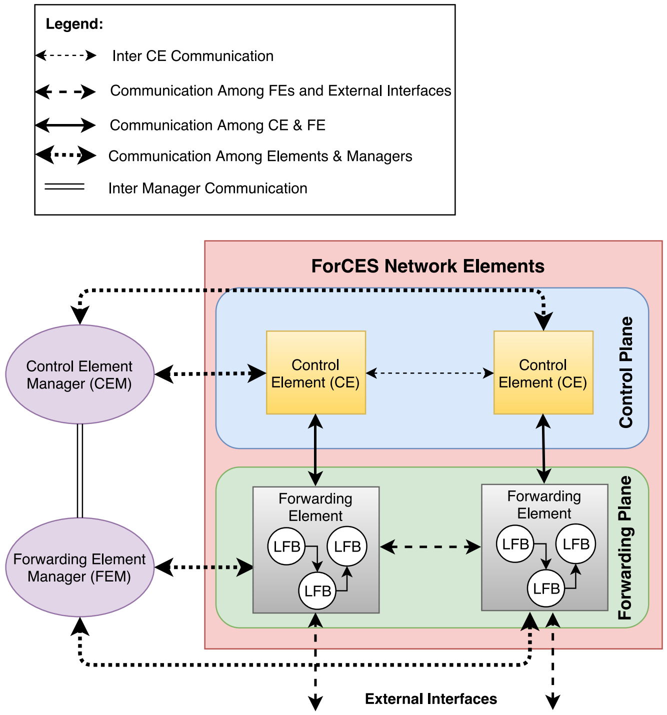

Legend:
Inter CE Communication
Communication Among FEs and External Interfaces
Communication Among CE & FE
Communication Among Elements & Managers
Inter Manager Communication
ForCES Network Elements
Control Element Manager (CEM)
Control Element (CE)
Control Element (CE)
Control Plane
Forwarding Element
LFB
LFB
LFB
Forwarding Element
LFB
LFB
LFB
Forwarding Plane
Forwarding Element Manager (FEM)
External Interfaces

Fig. 8. ForCES (Haleplidis et al., 2015) architecture.

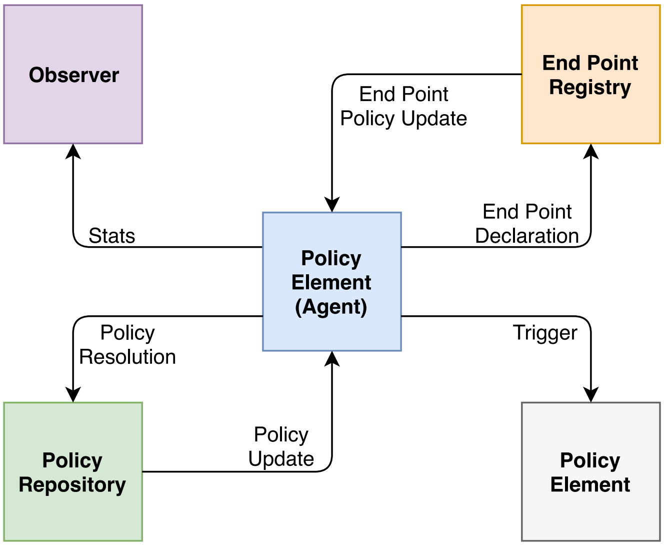


Fig. 9. OpFlex (Smith et al., 2020) architecture.

It uses a Logical Function Block (LFB) which resides inside FE and has a specific function to process packets and allows CE to control FE. FEs take LFBs as a graph and uses them to perform well-defined actions and do logical computations on packets that are passing through them. Each LFB can perform a single action on a packet.

ForCES messages are the key enabler to provide the control of FEs to CEs, and just like OpenFlow it also requires a transport protocol. This transport protocol not only provides communication between FE and CE but also provides extra services like reliability and security mechanisms. Rather than using TCP for this purpose (as used in OpenFlow), ForCES uses Stream Controlled Transmission Protocol (SCTP) (Stewart, 2007) which provides a range of reliability levels. Another major reason for using SCTP is the duplication and re-transmission nature of TCP, which in case of congestion, will make things worse. SCTP is also a good design choice for ForCES for its resiliency to failure detection with built-in recovery mechanisms.

A detailed comparison between OpenFlow and ForCES is discussed in (Hares, 2020) but in this paper, we provide a summarized and condensed comparison between these two competitors. Table 2 summarizes some of the major differences in OpenFlow and ForCES. One of the major differences between OpenFlow and ForCES is that every time new functionality is added, OpenFlow has to be modeled and standardized accordingly. Whereas, ForCES provides extensibility without the need for standardizing again and again. ForCES deployment is not restricted to any specific design of forwarding elements, but for OpenFlow, there are switch specifications of predefined features. However, despite being a mature solution, ForCES could not gain widespread adoption by vendors.

OpFlex (Smith et al., 2020), proposed through a draft for IETF from Cisco, is a protocol that provides communication between the centralized controller and data plane but with a very different scope as compared to OpenFlow. OpFlex is based on a declarative policy information model that means it centralizes only policy management and implementation but distributes intelligence and control. With the aim of scalability, OpFlex tries to distribute the complexity in such a way that forwarding devices are responsible for managing the whole network except policies. These policies are defined at logically centralized Policy Repository (PR) which communicates with Policy Elements (PE) using the OpFlex protocol. End Point devices are connected with Policy Elements and get registered by using the EndPoint Registry which is responsible for the addition/removal of End Points. Another repository in OpFlex is Observer (OB) which is responsible for statistics of faults and events. This interaction is depicted in Fig. 9. One major limitation of OpFlex, as compared to OpenFlow, is that it takes away the key feature of programming the network from a centralized controller.

NetConf (Enns et al., 2011) which uses Remote Procedure Call (RPC) paradigm, is a protocol that defines a simple mechanism by which network devices can be managed, configuration data can be retrieved, and new configuration data can be uploaded and manipulated. One of the key aspects of NetConf is that it closely mirrors the functionality of the management protocol to the native functionality of the device which directly reduces cost and allows timely access to new features. This proposal existed before SDN, but just like OpenFlow it also provides a straightforward API. This API can be used by applications to send and receive full or partial configuration datasets.

Segment routing allows the addition of state information to packet headers. This reduces the configuration overhead at nodes and results in a faster and simpler service setup. SRv6 (Ventre et al., 2018) is a similar SDN based segment routing approach for IPv6, while employing gRPC, REST, NETCONF, and SSH/CLI. The configuration of SRv6 rules in the devices is decomposed into two parts. One is the communication protocol which is used to send the requests to the SRv6 manager running at the node. The other part helps in the local configuration of the rules which uses the controller requests sent to SRv6 manager. Unlike OpenFlow, the controller does not interact with all the edge and core switches for topology discovery and flow rule installation, rather it uses OSPF v3 for topology discovery. The primary benefit is simplified configuration.

Insights: It is interesting to note that although ForCES was supported by IETF, it did not gain the industry confidence. The major reason was the vast deployment of and support from multiple developers and hardware vendors. Table 3 presents a summary of all the OpenFlow independent proposals with their objectives, solutions, and benefits. Some major benefits of ForCES over OpenFlow are its extensibility as well as no restriction on device specifications. It offers a rich set of features but lacks in open source support. Similarly, OpFlex restricts programming in networks which is a key feature of SDN. NetConf, on the other hand, is not a purpose-built interface for SDN, thus it does not provide enough flexibility. SRv6 minimizes the configuration in seg-

Table 2
Summary of comparison between OpenFlow and ForCES.

Southbound Interface	Standardizing Body	Protocol Used	Determinants	Extensibility	IP Versions Support
ForCES (Haleplidis et al., 2015)	IETF (Internet Engineering Task Force, 2020)	SCTP	Logical Functional Block	Yes	IPv4
OpenFlow (McKeown et al., 2008)	ONF (ONF, 2020)	TCP	Match Fields and Actions	No	IPv4 and IPv6

Table 3
Summary of OpenFlow independent SBI proposals.

Literature	Objective	Solution	Benefits
ForCES (Haleplidis et al., 2015)	Separation of Control and Forwarding Element in same Network Element	Logical Function Block	Enables Data and Control Plane separation using traditional network elements
OpFlex (Smith et al., 2020)	Distribute Complexity and Improved Scalability	Declarative Policy Model	Enhanced Scalability
NetConf (Enns et al., 2011)	Reduced Complexity and Enhance Performance	Remote Procedure Call (RPC)	Close to native functionality of switch and reduces cost
SRv6 (Ventre et al., 2018)	Minimize Configuration Overhead	Coexistance of Multiple Protocols	Provides faster and simpler configuration

ment routing for IPv6, however, it is not directly comparable to OpenFlow protocol.

3.4. Southbound APIs and perception plane

Internet of Things and wireless sensor networks are connected at the edge of a network and are considered *beyond vSwitches*. From the SDN perspective, they fall in the perception plane as shown in Fig. 3. Perception plane can be considered as an extension of data plane where devices are not completely OF compliant, hence many solutions have tried to extend OF to such devices. Here we classify these solutions for sensor networks and IoT networks.

3.4.1. SBI for wireless sensor networks

The development of smart sensors has enabled the monitoring of physical and environmental factors in numerous use cases. Software Defined Wireless Sensor Networking (SDWSN) (Kobo et al., 2017) is a relatively new paradigm for Low Rate-Wireless Personal Area Network which can be realized by infusing the SDN model into WSNs. Southbound Interface plays an integral role in SDN, but it is very hard to implement in SDWSN because of the following basic reasons:

- Matching fields in OpenFlow address centric, and flow entries are installed using the source IP and destination IP. Whereas, WSNs is data-centric, where data acquisition is more important than the source of data. Hence, flow creation is challenging in WSNs.
- Addressing in WSNs is not IP-based which prevents SDWSN SBI from creating flow entries. Moreover, it becomes hard to establish a TCP/IP based secure channel in SDWSN.
- To install or uninstall flows on sensor devices that are limited in size and memory, it may introduce overhead on the communication channel.
- Due to the device constraints, routing algorithms of WSNs are quite different from data centers or other networks. Hence, the topological information needed is more detailed and may not be represented by OpenFlow headers.

Sensor OpenFlow (SOF) (Luo et al., 2012) is based on standard OpenFlow but modified to the requirements of low capacity sensor nodes. It addresses challenges like; flow creation, secure channel between the control plane and data plane, control traffic overhead, and in-network processing, etc. To install the flows on sensor network devices, it redefines the flow tables due to special addressing schemes of WSNs. Flow tables are categorized into two classes: Class1 contains compact network unique addresses as 16-bit addresses in ZigBee, and Class2 uses concatenated attribute-value pairs. Class1 is handled by the use of OpenFlow eXtensible Match (OXM), a TLV format by adding two new addresses as OXM_SOF_Source and OXM_SOF_Destination. Another solution for this problem is to use *uIP* and *uIPv6*. *uIP* is an implementation of IPv4 in the Contiki operating system normally used for WSNs and the Internet of Things. Just like an OpenFlow secure channel, SOF suggests either to use Transport Protocol directly in WSNs or channels that can be supplied through *uIP or uIPv6*. To curb the control traffic between data and control planes, it proposed a customized solution of Control-Message Quenching (CMQ). However, the main focus was on the message type, packet format, and operations, hence authors did not provide any performance evaluation.

Software Defined Wireless Networks (SDWNs) (Costanzo et al., 2012) proposed some significant features to reduce energy consumption in WSNs by introducing duty cycles and in-network data aggregation. Another significant feature of SDWNs is to support the flexible definition of rules. Duty cycles are used to reduce energy consumption by turning the radio off when it is not being used. Another approach used to reduce energy consumption is in-network data aggregation. Unlike traditional OpenFlow, flexible flow entries are required for SDWNs because of its nature. SDWNs protocol architecture uses generic nodes as well as a sink node. All generic nodes run physical and MAC layer functionalities. The forwarding layer which is on top of the MAC layer is responsible to treat a packet as specified by the controller. All the generic nodes are connected to sink node(s) which has the same architecture as generic nodes except a few functionalities. A sink node has more computational and communication capabilities. Therefore, sinks are executed in Linux based embedded system. Embedded systems and sinks are connected through USB, RS232, or other interfaces. Another feature of the sink is to use a virtualizer with the responsibility of collecting information about generic nodes to build a detailed representation of network topology. Same as SOF, SDWNs also did not provide any performance evaluation and mainly focused on architectural details. Hence, actual performance is still unknown and maybe a research direction for the community.

SDN for WIreless SEnsors (SDN WISE) (Galluccio et al., 2015) goes one step further as compared to previous studies, and is implemented in OMNet++ with the objectives of reducing communication among sensor nodes to/from SDN controller and making sensor nodes programmable as a finite state machine, unlike standard OpenFlow which is stateless. In the control layer, it uses WISE Visor, which has Topology Manager for the collection of local information from the nodes and forwards it to the controller in the form of a graph with the topographic information, energy levels, and SNR of nodes. In the data plane, In-Networking Packet Processor is responsible for data aggregation and other in-network processing to reduce the overhead. Between control and data plane, there is an adaptation layer which is responsible for formatting the messages received from sinks in such a way that they can be handled by WISE Visor and vice versa. An application of SDN WISE is in (Anadiotis et al., 2018) where a unified system is realized, which enables communication of heterogeneous devices under a single network operating system, by adding subsystems like Sensor Node, Sensor Flow Rules, and Sensor Packet Subsystem.

A recent proposal of HUBs-Flow (Cicioğlu and ÇalhanHubsflow, 2019) is a southbound interface for SDN based Wireless Body Area Network (SDN-WBAN) which supports IEEE 802.15.6. This technique uses Hello and Echo control messages between the control plane and Hubs (which connect to body sensors and collect data), to piggyback information such as addresses, slot information, packet size, and packet priorities. Using this information, the SD-WBAN controller installs flow rules on HUBs. Similar to SDN-WISE, HUBsFlow also supports the duty cycle approach.

Insights: Table 4 summarizes the feature based comparison of different southbound interfaces for WSNs proposals. SOF and SDWN provide theoretical details, whereas SDN-WISE and HUBsFlow provided practical implementations. Nodes in sensor networks are susceptible to the movement which can cause path variation during packet transmission. It is very important to manage and monitor the movement of different nodes. One of the major challenges of SDN is to handle the effect of nodes entering or leaving the network. Another challenge is to build paths using different metrics (i.e. node energy and capability). The interface in such cases should be able to optimally gather required information for the controller. Moreover, controller placement in case of a wireless southbound interface is another research area (Dvir et al., 2019).

3.4.2. *SBI for Internet of Things*

Smart cities, smart grids, and intelligent transportation has expanded the Internet of Things domain significantly. IoT networks are not just sensor networks, rather they are more complex and implement WSNs as a sub-part of the whole ecosystem. Due to a large number of devices connected to the Internet, there are several challenges in IoT: scalability, connectivity, big data, security, and heterogeneity, etc. SDN provides a centralized controller and high-level management which hides the complexity to provide solutions for above-discussed problems. Implementation of SDN in IoT networks resolves several issues but introduces some new challenges.

- **Device Heterogeneity:** IoT devices are very diverse in nature and may use different types of technologies. Their capabilities also may vary, which requires new types of software-defined solutions, including controllers, virtual switches or SDGateways, and southbound interfaces.
- **Interface and Topological Diversity:** Each IoT device may have multiple communication technologies, e.g. WiFi, BLE, 5G, etc. As the flow installation on such a network is not simple, hence the southbound interfaces have to adapt. Moreover, SBIs also should be able to work with hybrid wired and multi-hop wireless networks.
- **Protocol Integration:** Each technology in IoT may have its packet format and processing rules. Flow installation with such a variety of protocols is a very challenging task.

To address these challenges, an OpenFlow like solution is required for IoT using SDN. There are several solutions available in the literature, but most of them do not resolve all the problems. Table 5 presents a summary of these proposals.

Salman et al. (2015) proposed architecture for implementing SDN in IoT and proposed a layered architecture to overcome problems like; scalability, big data, heterogeneity, and security. The bottommost layer is a device layer with different IoT devices and identifiers to differentiate them. The network layer is used to overcome the heterogeneity by using Software Defined Gateways (SD Gateways). SD Gateways can communicate with IoT devices using different technologies. An extension to OpenFlow is recommended but no specifications are discussed. However, for configuration purposes, some management protocols (e.g. NetConf, OF-Config, and Yang) are recommended. Another main feature of these SD Gateways is to reduce the power consumption because of big data problems. The control layer consists of SDN controllers with the responsibility of collecting topology information, path calculation, and forwarding rules. Security rules are defined using algorithms but no details on flow rule installation are discussed. At the topmost layer, there are different network applications.

Li et al. (2016) address the issues of interoperability, resource sharing, and flexibility for applications and services. It proposes a layered architecture where IoT devices are at the bottom layer and are referred to as the device layer. These devices are connected to switches or gateways which are at the communication layer. A module named Data Processing and Storage Center is also in the communication layer and controlled by the SDN controller. This module is capable of storing selective data of IoT devices and sinks, and also responsible for data format conversion. On top of the communication layer, there is a computing layer where SDN controllers are placed. The service layer is the topmost layer. Besides the data forwarding capability of switches and gateways, they can also store or cache local data and process it under the instruction of the SDN controller. An extension in OpenFlow version 1.0 is done by adding two flags. These flags mark data format and caching capabilities of the switch.

To resolve the problems of scalability and mobility, Ojo et al. (2016) provide a general architecture for IoT with the coupling of SDN and NFV. This architecture contains four layers; perception layer, data layer, control layer, and application layer. Devices in perception layer sense data and forward to data layer by using Software Defined enabled gateways. These Software Defined enabled gateways to provide management flexibility, as underlying devices belong to different technologies. Apart from these Software Defined gateways, there are also switches in the data plane. These devices can be programmed through controllers (e.g. ONOS, OpenDaylight) by using a southbound interface (e.g. OpenFlow, OvSDB, NetConf, BGP, etc.). This study lacks implementations and does not elaborate on how OF will be used beyond the gateway into the perception plane.

Multi-network INformation Architecture (MINA) (Qin et al., 2014) is another method to resolve the issues of heterogeneity and interoperability. It proposes a controller architecture and an OpenFlow like protocol. In the controller, it uses data collection components that collect network information and stores it in databases. This information is then utilized by other components of the controller. Among these components, there is an admin/analyst API which allows governing different control processes by controller itself as well as external programs. Other components are; task-resource matching, service solution specification, and flow scheduling. A task can be realized by a single service or multiple services. Task-resource matching specifies, which devices or applications can be used to complete a particular task. After matching, the controller maps the characteristics of devices and services involved in that matching by using the service solution specification component. It also handles specific requirements for devices or application constraints. These requirements are taken by the flow scheduling component to schedule flows. This component uses an algorithm to resolve the complexity due to the heterogeneity of different technologies. An OpenFlow like protocol is used in the communication layer for flow scheduling and data collection purposes. However, detailed discussion

Table 4
Summary of SBI proposals for wireless sensor networks.

Features	SOF (Luo et al., 2012)	SDWN (Costanzo et al., 2012)	SDN-WISE (Galluccio et al., 2015)	HUBsFlow (Cicioğlu and ÇalhanHubsflow, 2019)
Flow Creation	Yes	Yes	Yes	Yes
Field Matching	Yes	Yes	Yes	Yes
Action	No	Yes	Yes	Yes
Statistics	No	No	Yes	Yes
Data Aggregation	No	Yes	Yes	No
In-Network Processing	Yes	Yes	Yes	No
Duty Cycles Reductions	No	Yes	Yes	Yes
Mobility Management	No	No	No	No
Multi-metric Paths	No	No	Partial	No
Implementation Available	No	No	Yes	Yes

and working of it are not discussed in the paper.

CENSOR (Conti et al. 2019) architecture aims at providing security and enhances the performance in heterogeneous IoT networks. It uses the IoT controller, an IoT agent, and a specialized software remote attestation component for security. The lowest plane in CENSOR is the object plane which consists of IoT devices. IoT agent is a small piece of software that runs in these devices. This software is executed in a Trusted Platform Module (TPM) and attested periodically. Above the object plane is the data plane which has OpenFlow enabled devices. Similar to the object plane, devices in the data plane are also attested which makes a hierarchy. However, the communication between the object plane and data plane or object to the control plane is not entirely defined. Extension of to objects maybe future research direction in this regard.

In (Desai et al., 2016) Desai et al. provided a framework where an OpenFlow Management Device is responsible to provide communication between IoT devices and OpenFlow enabled switches. This device runs its own Linux based operating system. The bottom layer consists of hardware and protocols and these components are the base of this device. The device is claimed to be extensible to more protocols. Above this layer are libraries that provide different functionalities, such as security, web connection, and SQL. The application framework layer is on top of libraries with the resource manager, location manager, activity manager, and above this there is an application layer. Data plane devices communicate with the control plane as well as with this device using the OpenFlow protocol. Flow installation mechanisms, in this study, are not discussed.

Insights: Table 5 gives a comparative analysis of the proposals discussed in this section. Most of the literature related to IoT has focused on two parts: Controller and API. We find that more preference is given to controller design and it is assumed that OpenFlow or something similar will be able to communicate with the devices. The objective of this paper is limited to APIs, hence we limit this section to those works which have elaborated (even in passing) on the SBIs. It is important to highlight the necessity of SBIs specific for IoT devices. OF was not designed for mobile low capacity heterogeneous IoT devices. Hence, firstly it is important to evaluate the effect of communication performance in such networks, and then perhaps a more lightweight and customized API can be developed targeted for IoT networks. In IoT, OpenFlow is not limited to controller and vSwitch. It has to extend its reach to IoT devices. Hence, solutions which go beyond the SDN gateways is an important research area. Similarly, the SBI also needs to offer functionality other than flow installation.

3.5. SBIs and security challenges

OpenFlow is predominantly the most used SBI, however, Transport Layer Security (TLS) is optional in its configuration, which makes network infrastructure vulnerable. TLS is considered a standardized protocol however, for various reasons it is still open to attacks, particularly Man-in-The-Middle (MiTM). To secure OF based switch to controller communication, Agborubere et al. (Agborubere and Sanchez-Velazquez, 2017) proposed enhancement in TLS by presenting the client's certificate to the server for authentication. The server may send a randomized re-verification request to the client along with a time frame. Moreover, the client's Hello message ID can be used for verification.

Kloti et al. (Klöti et al., 2013) presents a security analysis of OpenFlow by using STRIDE (Hernan et al., 2006) vulnerability modeling technique. It shows that Denial of Service (DoS) and Information Disclosure attacks can be easily launched against OF devices. During a DoS attack, rate-limiting techniques are recommended which allow the controller to remain responsive along with event filtering which increases system resilience. Flow aggregation which is a proactive approach can also resolve DoS by making flow tables less prone to overflows. Attack detection can also be used as a controller application. Information disclosure attacks expose the state and services of the network. Proactive strategies are recommended to overcome this issue, however, this may create unnecessary overhead. Samociuk (Samociuk, 2015) presents another study which analyzes the pros, cons, usability, and implementation of TLS, Secure SHell (SSH), and IPSec protocols as a secure medium for OpenFlow communication between data plane and control plane.

Insights: Communication between data plane and control plane is the main functionality of SDN, therefore making this communication secure is fundamental. Since TLS protocol is not mandatory, all the SDN solution vendors may not implement it. At the same time, many vulnerabilities may still be hidden due to a lack of TLS based deployment. Hence, large scale testbeds for experimental evaluation of security may reveal more information.

4. Northbound interfaces (NBI) in SDN

Northbound Interface is one of the key pillars of SDN, as it provides programming abstraction for networks. It acts as a bridge between the control and management plane and provides a high-level abstraction for application development. The application development in the management plane is not as easy as it should be, and the main reason for it is the lack of standardization of the northbound interface. Unlike OpenFlow, there is no single API or protocol which different developers/vendors can use. One reason for this lack of standardization is the variation in applications and their requirements. Northbound Interface Work Group (NBIWG) (NBIWG, 2020) is an initiative of ONF (ONF, 2020), which was established for standardization purposes. However, it has witnessed little success, and because of this some controllers (e.g. Onix (Koponen et al., 2010), PANE (Ferguson et al., 2013), etc.) provide high-level APIs for their application development in SDN. Furthermore, programs and most of the controllers usually use REST API as Northbound Interface.

The working of SBIs is more like a protocol for communication, whereas NBIs are used for different objectives (Chen and Wu, 2017). identifies some key properties of NBIs. Using it as a rudimentary guideline, we group the literature of northbound interfaces on the following properties: portability, programmability, controller based, intent-based,

Table 5
Summary of proposals for Internet of Things.

Literature	Objective	Solution	Benefits	SBI Used	Changes in SBI	Limitations
Salman et al. (Salman et al., 2015)	Issues of scalability, reliability and heterogeneity	Implementation of gateways running genius algorithm	Provides control on IoT devices by extending OpenFlow	OpenFlow	Not discussed	No implementation available. Extensions in OpenFlow are not discussed properly
Li et al. (Li et al., 2016)	Resource sharing and interoperability	Data processing and storage center	Supports multiple services and interoperability	OpenFlow	Addition of two fields in OpenFlow header	Security designs are not focused
Ojo et al. (Ojo et al., 2016)	Scalability and Mobility	Use of Software Defined Gateways instead of traditional Gateways	Enhances network efficiency and agility	OpenFlow, OvSDB, BGP, PCEP, NetConf	Not discussed	No implementations available. Extensions in southbound protocols are not discussed
Qin et al. (Qin et al., 2014)	Heterogeneous nature of different technologies and Interoperability	IoT Multi-network Controller	Provides flexible, effective and efficient management	OpenFlow like Protocol	Not discussed	Lacks in details of southbound protocol
CENSOR (Conti et al. 2019)	Enhance security and performance	IoT agents and controller with remote attestation component	Improves performance and security	OpenFlow	Not discussed	No discussion about flow installation
Desai et al. (Desai et al., 2016)	Better Control on IoT Heterogeneous Devices	Introduced OpenFlow enabled management device using Linux kernal	Supports heterogeneous IoT devices to communicate with Remote Processing systems in cloud	OpenFlow	Not discussed	No discussion about flow installation

and virtualization.

Portability provides low-level abstraction and is used to resolve the compatibility issues among different versions of OpenFlow or other southbound APIs and different hardware. It provides the guarantee of correct packet processing on a wide range of data plane devices. Programmability refers to the use of high-level programming languages or dedicated languages for SDN. By using high-level languages, networks can be configured for the services required, which is referred to as prescribed usage. Whereas, there is an intent-based usage of NBIs, which is opposite to prescribed usage. In this model, application requirements are described in natural language and the controller is intelligent enough to integrate desired services with its core functionality.

Due to the absence of a standardized northbound interface, many of the controllers use ad-hoc APIs. Whereas, some of the controllers proposed their high-level interfaces referred to as controller-based APIs. Moreover, some of the controllers also support intents where high-level policies can be declared. Interfaces allow the sharing of resources of underlying network devices in virtualization and reduce capital and operational costs. Due to the blurriness of virtualization techniques in SDN, we have discussed virtualization as a separate section. In this section, we discuss portability, programmability, controller based, and intent-based NBIs.

4.1. Portability in NBI

A reason which hinders application development is a gap between hardware vendors and application developers which reduces portability i.e. guarantee of correct packet processing and performance over a wide range of network switches. There are a large number of different (hardware and software) switching products available from dozens of vendors, which differ in the data plane, switch-controller interaction, and fixed/flexible pipeline (Banks, 2020).

Software Friendly Networking (SFNet) (Yap et al., 2010) provides an interface between the control and application planes of the SDN paradigm with a role to hide the lower network protocols from the application. It is a high-level API and directly interacts with the underlying network. It translates application requirements and programs network accordingly to provide services. It uses a JSON file to send information requests and congestion reports from the network to find a better path among hosts. In case of congestion, it allows applications to back off if they choose to do so. It can be beneficial in case of delay sensitive traffic. It also supports bandwidth reservations and grants/denies the request depending upon the availability of requested bandwidth. This requirement of bandwidth can be prioritized for video on demand or Voice over IP (VoIP) traffic. It also supports multicasting to a set of IP addresses participating in it.

NOSIX (Yu et al., 2014) addresses the problem of a diverse range of underlying switches to enhance performance. It proposes the use of Virtual Flow Table (VFT) which is a basic component used by applications to freely define the rules without any concern of delays and throughput of updates and notifications. It allows applications to predefine the VFTs pipeline and then install rules in these tables. The predefinition of a pipeline is allowed because it is difficult to perform dynamic reconfiguration of a physical pipeline of switches. Moreover, the rules in VFTs do not need to be always in the physical flow tables of a switch. Switch drivers, on the other hand, maps the VFT pipeline onto a physical pipeline available on the switch. The switch driver can be placed either at the lower layer of the controller or on the switch itself. However, it is beneficial to place switch drivers at the lower stack of the controller because it is easy to program a software-based switch-driver at the controller than a switch. In NOSIX, control applications can be written as a pipeline of VFTs and vendor-supplied drivers then transform them into switch configuration.

Another solution is tinyNBI (Casey et al., 2014) which is a language-independent solution and resolves portability issues in terms of different OpenFlow versions and specifications. It is designed in C language, and provides a complete set of OpenFlow semantics and handles multiple versions of OpenFlow and variable switch capabilities without requiring any additional efforts. The main purpose of this NBI is to provide a foundational interface for application development. It uses a data model that makes a clear distinction between control and data plane abstractions. Every abstraction has three components; configuration, capabilities, and statistics. The configuration is modifiable data by the interface. Capabilities describe the behavior of abstractions and it is a non-modifiable state. Whereas, statistics are read-only data and describes how abstraction has behaved. All the abstractions are not present in all OpenFlow versions (e.g. Meter Table, Group Table). Abstractions that are not available are handled in three ways; seamless emulation, switch offloading and error indication. Seamless emulation defines the abstraction but with limited capabilities. Switch offloading represents offloading of missing switch capabilities to the controller. In some scenarios, it is not possible to provide missing behavior which provides an error indication. It is not a high-level interface and does not provide information like network topology, switching, routing, or load balancing. Instead, it provides independence from the ever-changing structure and semantics of OpenFlow and provides maximum portability and re-use potential.

Insights: One of the major responsibilities of NBI is to provide information on underlying devices to developers. However, there is a diverse range of underlying devices and southbound protocols. Portability provides solutions for compatibility issues of this diversity of network elements and protocols. Table 6 presents a summary of different proposals for portability in NBIs. These solutions focus on either data plane devices or protocols, but a single solution for both of these is still an open research challenge.

4.2. Programmability of NBI

Similar to application development, which shifted from assembly language to high-level languages like Python and Java, network languages have also shifted from low-level language (e.g. low-level language used in OpenFlow) to high level (e.g. Frenetic, Procera, etc.). These high-level languages in SDN provide flexibility in network management and make it less error-prone.

Current network elements often perform multiple tasks simultaneously (e.g. routing, monitoring, access control, etc.). However, decoupling these tasks is almost impossible, because packet handling rules installed by one can conflict with another. OpenFlow interface is defined at a very low level of abstraction which directs capabilities of the switch hardware. To apply high-level concepts, programmers can not directly use the OpenFlow instruction set. Another issue with OpenFlow is that, packets are processed by the controller if the switch cannot process them due to lack of flow information, hence programmers have to perform two-tier programming, one for the packets being processed at the controller and other which need to be processed at the switch level. The literature related to NBI programming languages has been discussed in the following sub-section. Before it, we have identified different properties of such languages.

4.2.1. NBI programming language feature classification

The feature classification of NBI-PL is shown in Fig. 10. Here we give a brief description of each feature.

Flow Installation refers to the way in which forwarding rules can be installed on switches: i.e. *Reactive* and *Proactive*. Almost all of the languages provide a reactive approach, however, some additionally provide proactive flow installation capabilities. In a reactive approach, when a new packet arrives at the switch, and no flow information is available, then this packet is forwarded to the controller, which based on its program logic, installs flows in the flow table of the switch. This method introduces latency as every new packet will be sent to the controller for flow installation. On the other hand, the proactive approach

Table 6
Summary of Portability solutions in NBI Proposals.

Literature	Objective	Solution	Benefits
SFNet (Yap et al., 2010)	Interaction among applications and underlying devices	Uses plug-ins for various applications	Allows bandwidth reservations, multi-casting and supports congestion inquiry
NOSIX (Yu et al., 2014)	Switch diversity and performance enhancement	Virtual Flow Table (VFT) and Switch Driver	Provides flexibility to programmers for a diverse landscape of data plane devices
tinyNBI (Casey et al., 2014)	Foundational interface for OpenFlow versions	Uses data model to abstract specifications	Supports multiple version of OpenFlow and provides extensibility

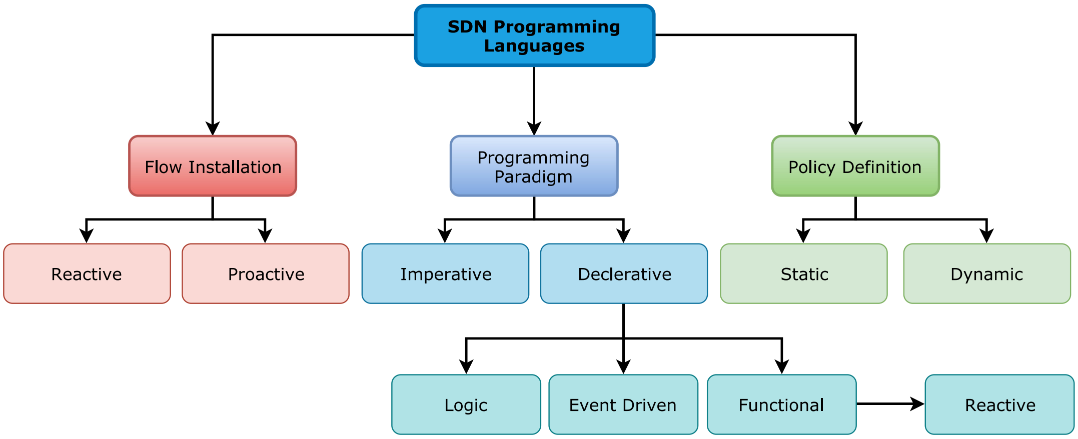


Fig. 10. Feature classification of NBI programming languages in SDN.

eliminates latency as every packet is not sent to the controller for forwarding rules. Predefined rules and actions are installed in flow tables. Languages that perform proactive installation, pre-compute forwarding tables for the whole network, and controller only modifies flow tables in case of link failure or other external events.

Policy Definition is a set of aggregated rules for the network, and each policy is a set of conditions and a corresponding set of actions where condition defines which policy rule will be applied. Policy definitions are classified into *static* and *dynamic* policies. Static policies (most traditional methods for firewall filters) are a predefined set of rules and actions. Dynamic policies, on the other hand, can be changed according to network conditions. Although most languages have dynamic capabilities, there are some which only work with the static policy definition.

Programming Paradigm reflects different ways of building the structure and elements of any program. There are two different paradigms in SDN programming: *imperative* and *declarative*. Imperative paradigm can be viewed as a traditional programming structure and allows the programmer to specify all the steps to solve any particular problem. In the declarative paradigm, one only needs to specify what program must do, not how to do it (Qadir and Hasan, 2015). The most known example of declarative paradigm is Structured Query Language (SQL), where a query is stated and the database engine executes it. The declarative paradigm has further three sub-paradigms: logic programming, event-driven, and functional programming. In *Logic Programming*, the compiler applies an algorithm that scans all possible combinations to a set of defined inference rules to postulates and resolves a query. *Event Driven Programming* allows the program to respond to any particular event. As soon as an event is received, automatic action is triggered. This action can either be some computation or trigger another event. *Functional Programming* acts as an evaluation of some mathematical functions and avoids state changes. The *Reactive Programming* in the declarative paradigm facilitates programs to react to external events. For example, a spreadsheet which typically has values or formulas in its cells. Whenever cell changes, formulas are recalculated automatically. A combination of functional and reactive programming makes *Functional Reactive Programming* (FRP) which models reactive behavior in functional languages. Programs in FRP correspond to mathematical functions in a declarative manner.

4.2.2. Programming languages for NBI

Here we present a list of programming languages specifically designed for SDN usage. Table 7 presents a listing of the classification features of these languages.

Frenetic (Foster et al., 2011), embedded in Python, proposes two levels of abstraction which includes a set of source level operators for constructing and manipulating streams of network traffic, and a declarative system that handles all the details of installing and un-installing low-level rules on switches. It provides a declarative solution with a modular design. Using Frenetic, programmers do not need to be concerned about flow rule installation which may prevent the controller from analyzing other traffic. Flow-based Management Language (FML) (Hinrichs et al., 2009) is a high-level declarative language based on data log for policy configuration about a verity of management tasks within a single framework for large enterprise networks. The main issue with FML is that it applies policy on all packets of any particular flow and does not provide much flexibility. Flog (Katta et al., 2020) is another event-driven and forward chaining language for SDN which combines the idea of FML and Frenetic as it follows logic programming technique as FML and factored programs in three components like Frenetic: a method to query network state, a component to process data generated after queries, and a mechanism to generate rules for installing on network elements. It uses event-driven and logic paradigms, and programs in this language execute upon the occurrence of an event in the network.

To design the network policies, a language must be expressive enough to capture these policies. Procera (Voellmy et al., 2012), as compared to FML, is more expressive and handles policies in a better manner. It allows network operators to define policies that react to dynamic changes in different network conditions. It provides an extensible, expressive, and compositional framework. An up-gradation of Frenetic is proposed by the same developers as Pyretic (Monsanto et al., 2013), which is a Python-based platform that allows application developers to design sophisticated applications. Same as Procera it also helps in policy-based application design. It tries to resolve the shortcoming in OpenFlow in terms of its programming nature and its role as a programming interface to switch in the network. Policies in Pyretic support modular programming and also facilitates the creation of dynamic policies.

NetCore (Monsanto et al., 2012) provides packet forwarding policies in SDN which is expressive, compositional, and has formal semantics. Rather than using the SDN controller, NetCore provides compilation algorithms and couples them with the run time system which issues flow installation rule commands. It exclusively focuses on flow tables simplicity but lacks expressiveness. To solve this issue, FlowLog (Nelson et al., 2014) which is a tier-less programming language, provides flexibility to use external full-featured libraries. Nettle (Voellmy and Hudak, 2011) using the mantra "*Don't Configure the Network, Program it!*" adopts ideas from Function Reactive Programming (FRP) and design methodology from Domain Specific Language (DSL), is embedded in strongly typed language Haskell (which works as a host language). It can be understood as signal functions used in electrical signals and provides flexibility to change these functions as well as retrieve discrete-time

Table 7
Feature based summery of SDN languages for NBIs.

Literature	Flow Installation		Programming		Description
Reactive	Proactive	Policy Definition	Paradigm		
Frenetic (Foster et al., 2011)	✓	–	D	FR	Designed to avoid race condition by using well defined high level programming abstraction
FML (Hinrichs et al., 2009)	✓	–	S	L	Designed for policy definition in enterprise networks
Flog (Katta et al.,)	✓	–	D	L/ED	Executes programs on event occurrence in network
Procera (Voellmy et al., 2012)	✓	–	D	FR	An expressive language and used for policy handling and provide an extensible and compositional framework
Pyretic (Monsanto et al., 2013)	✓	✓	D	I	An upgrade of Frentic and also used for policy handling in a transparent framework
Nettle (Voellmy and Hudak, 2011)	✓	✓	D	FR	Allows programmers to deal with streams instead of events and can be understood as signal functions
NetKAT (Anderson et al., 2014)	✓	✓	D	F	Provides reasoning for network mapping and traffic isolation using Kleene Algebra and Tests
NetCore (Monsanto et al., 2012)	✓	✓	D	FR	Provides means for packet forwarding policies and generate flow installation commands
FlowLog (Nelson et al., 2014)	✓	✓	D	F	Allows programmers to use external full featured libraries
FatTire (Reitblatt et al., 2013)	✓	✓	D	F	Provides fault tolerance in the networks and describe network paths
Kinetic (Kim et al., 2015)	✓	✓	D	ED	Domain Specific Language that allows to control the network dynamically
Merlin (Soulé et al., 2013)	✓	–	D	F	Delegates sub-policies to different tenants and allow them to modify according to their requirements

Legend: D = Dynamic, S=Static, I=Imperative, ED = Event Driven, FR=Functional Reactive, L = Logic, F=Functional.

and contentious time values.

Another proposal for network programming in SDN is NetKAT (Anderson et al., 2014) which uses a mathematical structure called Kleene Algebra for tests and provides solid mathematical semantic foundations. It provides an equational theory for reachability, traffic isolation and compiler correctness of algorithms.

FatTire (Reitblatt et al., 2013) provides forwarding and fault tolerance policies. It allows programmers to set legal paths through the network along with the fault tolerance of those paths. Network conditions are always changing and operators have to change the configuration manually. Kinetic (Kim et al., 2015), based on Pyretic, proposed an intuitive mechanism to change these configurations dynamically. It expresses network policy as a Finite State Machine (FSM) which captures dynamics and amenable to verifications. It also verifies the correctness of these high-level specifications.

To manage the networks, administrators need to configure their network very carefully because the misconfiguration of a single device may bring an undesired behavior to the whole network. By using Merlin (Soulé et al., 2013), administrators can express policies in a high-level declarative language. Merlin compiler uses program partitioning to transform global policies to smaller sub-policies which are distributed to different components of the network automatically, and delegates these sub-policies to different tenants who can modify them to reflect their custom requirements.

To provide QoS in traffic routing using SDN and Northbound Interface, Software-Defined Constrained programming Routing (SCOR) (Layeghy et al., 2016) is another solution that is based on Constrained Programming (CP). SCOR divides NBI into two layers: the upper layer is *CP Based Programming Language*, and the lower layer is *QoS Routing and Traffic Engineering Interface*. The lower layer is defined to address the requirements and has nine predicates: i.e. network path, capacity guarantee, delay, path cost, etc. SCOR is implemented in MiniZinc (Nethercote et al., 2007) which is declarative constrained programming.

4.2.3. *Insights on NBI programmability*

SDN languages are evolving to enhance the abstractions for programmability in networks. Several SDN languages can take advantage of some new features of OpenFlow. However, it requires adding new libraries and active support from the research and development community. Currently, SDN languages have limited libraries as well as community-based contributions, which can be an active area for development.

4.3. *Controller-based and intent-based NBIs*

The absence of a standard northbound interface (or any framework for it) has allowed different vendors to package their customized solutions for their controller products. Some of the controllers provide their high-level interface, whereas some use ad-hoc APIs. In this subsection, we highlight four NBIs which are specific to controllers. The objective is not to discuss the controller itself, but rather dissect the NBI solution. Out of these four, two (Onix and PANE) proposed their high-level API, whereas the two most popular controllers (OpenDaylight and ONOS) use multiple APIs for different northbound functionalities. In addition, we discuss a new type of NBI which is based on intent (policy rather than the specification) of the application.

Controller-based NBIs: Participatory Networking (PANE) (Ferguson et al., 2013) is an example of an SDN controller, which provides its high-level API. This API between the control plane and applications allows reading the current state of the network and writing configuration. It involves two major issues; 1) decompose the control and visibility of the network, and 2) resolve the conflicts between different participants. To solve these issues, it uses three types of messages; requests, queries, and hints. Request messages are used for network resources (e.g. bandwidth and access control). A request may affect the state of the network for a time interval. Queries are used to read the network state (e.g. traffic between hosts and bandwidth available). Hints provide the network information which may help to improve the network performance. Moreover, it provides an API where end-host applications can dynamically request network resources (e.g. bandwidth reservation). To avoid starvation and exceeding the bandwidth limits set by the administrator, it uses a verification engine. Although very useful, the effect of excessive requests is not addressed in this work.

Onix (Koponen et al., 2010) is another example of a controller that provides its northbound interface. It defines a general API which enables scalable application development. It also allows control applications to read and write the state of network elements. Moreover, it uses a data-centric approach that provides consistency between control applications and underlying network devices. It consists of a data model, representing network infrastructure, where each network element corresponds to one or more data objects. Control logic reads the current state associated with a particular object, operates on this object to alter the state, and registers notifications for state changes on this object. A copy of these notifications and changes are also placed in Network Information Base (NIB). Detailed discussion on NIB is given in Section VI.

ONOS (Berde et al., 2014) is one of the most popular open source SDN controllers which is driven by OpenNetworks Laboratory (ONLab) founded in 2012. ONOS mainly focuses on scalability, high performance, resilience, and next-generation device support. It uses a collection of Open Services Gateway initiative (OSGi) bundles and provides interaction with applications by using Java and REST APIs. It supports both command line and graphical user interface to provide flexibility in application development and network administration through REST API. It also provides a wide range of templates for the development of new applications. Due to the distributed nature of this controller, it uses general-purpose Remote Procedure Call (gRPC) (gRPC, 2020) which simplifies the creation of distributed applications. In gRPC, methods of a client application can be called directly on server application, running on a different machine, as it was a local object.

Another open source project is OpenDaylight (2020), a founding member of Linux Foundation Networking (LFN), which is widely supported by industry and research community. This controller project was started in 2013 and written in Java with a focus on network programmability. OpenDaylight also uses OSGi bundles which run as Apache Karaf (Karaf, 2020) components. DLUX (DLUX, 2020) in OpenDaylight is used as a web-based interface and represents several features including a graphical user interface for topology representation. Most of the interfaces can be visualized through Yang-User Interface (YangUI, 2020). Yang-UI is a collection of REST APIs which enable developers to query network information as well as configure it. For example, the network topology component in Yang-UI provides comprehensive information about the whole network, and the inventory component provides detailed information about statistics.

Comparing ONOS and ODL is interesting as both are written in Java, however, they have significant differences. From the controller's perspective, the main focus of ODL is to bring legacy (BGP, SNMP, etc.) and next-generation networks together. whereas, ONOS focuses on enhancing the performance of the network. Both of them offer GUI, however, ONOS has a cleaner presentation style. In terms of scalability, ODL can scale up to 400 switches but ONOS is not able to scale to this extent. Works in (Zhu et al., 2019) & (Badotra and Panda, 2019), present a detailed comparative analysis of these controllers. Although (Zhu et al., 2019) includes many other controllers also, both these works conclude that ODL is more feature-rich and performs better.

Intent-based NBIs: The adoption of SDN critically depends on its ability to support multiple types of applications through NBI. Most of the solutions for NBI are ad-hoc, vendor specific and have limited capabilities. The intent-based model attempts to resolve these issues and allows the declaration of high-level policies, instead of a detailed specification of different networking mechanisms. The above-mentioned ONOS and OpenDaylight controllers also support intent-based northbound interfaces.

The intent framework of ONOS allows applications to specify their requirement of network control in the form of policies instead of mechanisms (Intent framework, 2020). These policy-based directives are referred to as Intent. These high-level intents are translated into installable forwarding rules which are essential operations to control network. Moreover, these intents can be identified by using two parameters; Application_ID and Intent_ID. Application_ID represents an application that creates a particular intent. Whereas, Intent_ID is generated whenever an intent is created. As soon as the intent is submitted by an application, it is directly sent to the compilation phase. This phase uses an intent compiler, which converts them into installable intents. If an application asks for an unavailable objective (e.g. connectivity among non-connected segments), this phase will see for an alternate approach to recompile. After compiling an intent, it is sent to the installing phase, where an intent installer is responsible to convert installable intents into flow rules. An intent manager provides coordination between intent provider and intent installer.

Network Intent Composition (NIC) (Rodrigues, 2020) is an internal project of OpenDaylight which is currently in the incubation state. NIC provides interaction between core modules of OpenDaylight or external applications to fulfill user desires. It uses current network service functions of OpenDaylight and southbound interfaces to control virtual and physical network elements. In OpenDaylight, a component referred to as *renderer* is used to transform the intents to the implementation of flow rules. A wide range of renderers is supported in various versions of OpenDaylight, which includes; Network Modeling (NEMO) renderer, OpenFlow renderer, Virtual Tenant Network (VTN) renderer, and Group-Based Policy (GBP) renderer. There are two core functions (i.e. hazelcast and MD-SAL) that supply the base models for NIC capability. On top of these functions, a renderer can be installed. This renderer transforms an intent using a particular project (e.g. VTN, NEMO, etc.) for network modifications. For example, the NEMO renderer is a feature that will transform an intent to a network modification by using NEMO project (NEMO, 2015,2020) in OpenDaylight.

To create new services as well as to compose or split current services of network applications, Pham et al. (Pham and Hoang, 2016) proposed a solution for intent-based NBI. The design principles of this study are data decentralization, web service components, process isolation, and robustness. Moreover, it proposed a three-tier architecture. To provide flexibility, these tiers work independently (i.e. every tier can change its components without affecting other tiers). The tiers are database tier, business logic tier, and presentation tier. Application states are stored in the database tier, which can be retrieved in different contexts. Service creation and composition is handled by the business tier. In this tier, a service registry is used to discover existing services whereas, new services can be created as atomic service. To integrate new and existing services, service integration element is used. By using CLI, REST, and programming interface, the presentation tier takes input from the user. To analyze the process it uses Domain-Driven Design (DDD) where requirements are decomposed into smaller problems and the solution for each problem is built. For example, composite intents are decomposed into based intents which are further decomposed into solution intents.

Insights: Due to a diverse range of controller-based NBIs, applications designed for one controller may not work for any other controller. However, it is a good initiative to have intent based NBIs but only a few controllers support it. Table 8 presents a summary of different controller based and intent-based NBIs, where OpenDaylight and ONOS support both controllers based and intent-based NBIs. PANE and Onix offer their own NBIs. Unlike the southbound interface, a mature and comprehensive solution for NBIs is still missing.

Table 8
Controller based and intent based NBIs.

Controller/Literature	NBI	Controller or Intent based	GUI	Security
OpenDaylight (OpenDaylight, 2020)	OSGi and REST APIs	Both	Yes	Strong
Onix (Koponen et al., 2010)	Onix API	Controller	No	Average
PANE (Ferguson et al., 2013)	PANE API	Controller	No	Weak
ONOS (Berde et al., 2014)	Java and REST APIs	Both	Yes	Strong
Pham et al. (Pham and Hoang, 2016)	Programming APIs, CLI and REST API	Intent	Yes	Weak

4.4. Security in northbound interfaces

The centralized architecture of SDN gives the controller a global view and access to the network, which requires that the controllers and northbound interfaces must be secured. A lack of security in these interfaces can allow applications to gain full access to the underlying network elements. However, few articles in literature address NBI security issues. Banse et al. (Banse and Rangarajan, 2015) proposed a web-based northbound interface that is not only secure but also controller independent and supports external applications. Applications require special permission each time they access or modify resources. The permissions of the messages can be varied. For example, an application can generate messages to create new flows but not for the configuration. Similarly, applications may require permission for different events like topology events, message events, and application events.

Some other challenges like unauthorized access of controller, illegal function calls, malicious flow entry injection, exhausting resources, and Man-in-The-Middle (MiTM) are addressed by ControllerSEPA (Tseng et al., 2016). To solve these issues authors suggested solutions like authentication, authorization, accounting, flow rule verification, application isolation, and application monitoring. Moreover, security enhancing plug-in for OpenFlow application is proposed which sets up a connection with controller and permission is delegated to ControllerSEPA. Applications can only communicate with the services which are provided by ControllerSEPA. To overcome information disclosure, ControllerSEPA repacks all the services of controllers and provides new APIs to applications that also solve controller specification problems.

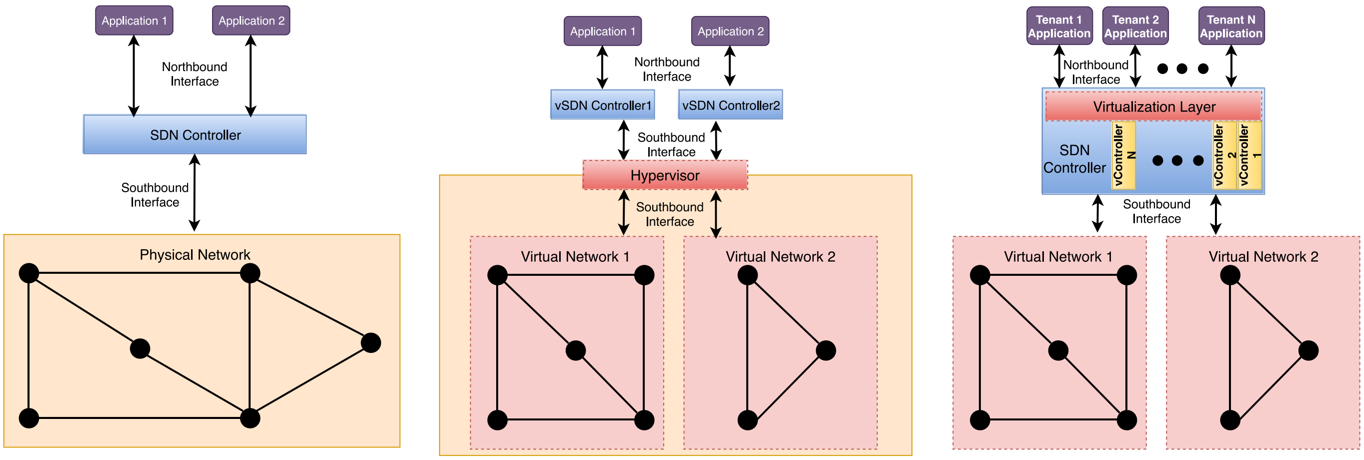
OpenDaylight in AAA project (OpenDaylight AAA project, 2020) uses a token that can be accessed after providing user name and password and later this token can be used by the application or services to access the network resources. But this approach only authenticates a user and not the application. Oktian et al. (Oktian, 2015) solves this issue by using OAuth 2.0 (Hardt, 2012) protocol which authenticates both user and application. In this approach, applications need to register with an authentication server before accessing the network. After registration, an API key and API secret are generated which are used to identify and verify the application. Similar to application registration, users can also be registered.

To avoid spoofing, disclosure of network resources, and tempering application to controller messages in SDN based Vehicular Ad hoc Networks (VANETs), BENBI (Weng et al., 2019) provides scalable and dynamic access control. It uses Identity based Broadcast Encryption where broadcaster transmits encrypted messages to all listeners. These listeners are appointed by the broadcaster and they only can decrypt the messages. Moreover, BENBI also supports multiple domains where different administrators agree on the same secret key to provide secure cross-domain communication.

Insights: The benefits of SDN control can be leveraged by third-party applications through northbound interfaces. Unfortunately, the implementation of these interfaces is not standardized nor are the security requirements for them. Hence, either the controllers only allow specific propitiatory applications to communicate with, or leaves it to the application designer to secure the communication. One promising direction can be to wrap the controller in a security layer, which can provide security for NBI communication as well as application access.

5. Virtualization and SDN interfaces

The objective of this section is to discuss the use of hypervisors and NFV techniques (which enable virtualization) and their effect on southbound or northbound interfaces. SDN, NFV, and hypervisors are highly complementary to each other and maximize network resource utilization. Fig. 11 represents this relationship between SDN interfaces and virtualization. Conventional SDN architecture is depicted in Fig. 11(a), whereas a hypervisor is placed at the southbound interface as shown in Fig. 11(b). In this scenario, a physical network is divided into multiple virtual networks to make slices, and different applications can run on the virtual SDN controller. Sometimes tenants do not require



(a) SDN Network

(b) Virtualization using Southbound Interface

(c) Virtualization using Northbound Interface

Fig. 11. SDN interfaces and virtualization.

a full-fledged controller. In this situation, virtualization schemes run as a module on the controller as shown in Fig. 11(c). Thus, different tenants can run their applications on a single controller using northbound interfaces.

It is important to note, that the objective is not to discuss virtualization for SDN, rather dissect the effect on interface communication. OpenFlow networks have the potential to open the control of network but only one user can work on network devices at a time. FlowVisor (Sherwood et al., 2010a, 2010b) allows multiple users to operate independently on their slices without any conflicts. FlowVisor acts as a transparent proxy where OpenFlow messages generated by network devices go to the FlowVisor from where they are routed to appropriate users. In this way, FlowVisor acts as a virtual controller for underlying switches and virtual switch for the users. It partitions the flow tables in so-called flow-spaces and as a result OpenFlow switches can be manipulated by multiple controllers. Furthermore, Auxiliary Software Datapath is applied to overcome the limitation of flow table size in OpenFlow switches.

AutoSlice (Bozakov et al., 2012) proposed another solution to the virtualization of underlying network devices with the main focus on scalability. It uses a hypervisor, placed between switch and controller, which can handle a large number of flow table control messages and multiple tenants. This hypervisor is composed of Management Module (MM) and multiple Control Proxies (CPX) which can evenly distribute the control load. Upon receiving a request, the Management Module determines the appropriate Virtual SDN (vSDN) it belongs to, and CPX installs flow entries accordingly.

OpenVirteX (Al-Shabibi et al., 2014) is an approach which provides topology virtualization, address virtualization, and control function virtualization. Tenants can request a topology by providing a mapping between the elements in physical topology and desired virtual topology. This virtual topology can either be an exact physical topology or a subgraph of it. It also grants permission to tenants for custom address assignments. Multiple addresses can cause problems at the time of flow installation. To resolve this problem OpenVirteX generates a globally unique tenant ID, to allow every tenant to run its network operating system which maps various control functions for the virtual network.

Another virtualization scheme for WAN is AutoVFlow (Yamanaka et al., 2014) which uses multiple controllers. It resides between switch and controllers and uses the southbound interface for virtualization. It allows delegation of configuration roles to multiple administrators and implements a mechanism of flow space virtualization on Wide-Area Network without any need for third party software as it is done in OpenVirteX (Al-Shabibi et al., 2014).

In some cases, a tenant does not require full-fledged control over the network. In such a situation, each tenant can use FlowN (Drutskoy et al., 2013) which resides inside a controller and uses the northbound interface. It allows tenants to run their applications over a single SDN controller. Authors in (Drutskoy et al., 2013) used NOX as an SDN controller to implement FlowN. Rather than running the controller for each tenant, they used a shared controller for all the tenants. Virtualization in FlowN is container-based where applications running on top of controller consist of handlers that respond to network events. Each of these applications has the illusion that they are running on their controller rather than a shared one.

Network Hypervisor (Huang and Griffioen, 2013) seamlessly handles the complexity of different levels of abstractions and a variety of APIs in SDN. Similar to FlowN, it also acts as a controller to the applications and provides visualization of the underlying network devices. By using this approach, SDN applications interact with Network Hypervisor through the northbound interface and compile the attributes of respective APIs. It is implemented on top GENI testbed and supports GENI API (Berman et al., 2014).

Network Virtualization Platform (NVP) (Koponen et al., 2014) uses Onix (Koponen et al., 2010) controller platform where cloud tenants can manage data center network resources by using OpenVSwitch (OVS). Rather than tenants running their controller, it provides a virtual slice to tenants to manage their resources by running their applications through high-level API. It resides inside the controller and uses a northbound interface for virtualization purposes.

libNetVirt (Turull et al., 2012) is an approach that is used for virtualization of networks in the same way as it can be done in machine virtualization. It is divided into two components; i) generic interface, and ii) drivers. Generic Interface is a set of functions that allows the interaction between virtual networks. Drivers, on the other hand, are the technology dependent elements. For this virtualization scheme northbound interface is involved. For the southbound interface, it is not dependent on OpenFlow, and another proposal can also be used.

Insights: Table 9 presents a summary of virtualization techniques along with the details of interfaces involved. Some proposals use hypervisor which is placed and works as a proxy between switches and controllers. Whereas, in some cases, a full-fledged controller is not required by tenants. In this case, the controller is divided into multiple virtual controllers and tenants can run their application by using their slice of the virtual controller.

6. East/westbound interface (E/WBI) in SDN

The main advantage of SDN is to provide a centralized view of the network, but the exponential increase in network devices has led to new challenges. On one hand, deploying a new SDN domain is a relatively simple approach, but to make this new domain interoperable with traditional Autonomous Systems (AS), is challenging. A common method for this purpose is to use BGP for information sharing. Similarly, PCEP (Vasseur and Roux, 2009) and GMPLS (Mannie, 2004) can also be used for communication among SDN controllers and legacy networks (Kreutz et al., 2015). However, these solutions are not designed for SDN, rather they are used as makeshift solutions. East/Westbound Interface is used for interconnection between different SDN domains or interaction between SDN and traditional network domains, where east refers to SDN-SDN communication, and west refers to legacy-SDN communication.

6.1. Interaction between SDN domains (eastbound APIs)

Centralized control of SDN makes network programmability & management simple, but for large scale networks, there are new challenges like scalability, security, and availability (Wibowo et al., 2017).

- Scalability: A single controller can manage only a limited number of switches, which causes scalability issues. In SDN architecture, a centralized controller is a key artifact, but it also creates a performance bottleneck as soon as the number of switches increases. It also introduces the risk of a single node failure problem.
- Security: A range of security problems may affect the SDN controller and its performance. In the case of a large scale network (Dhawan et al., 2015), a Denial of Service (DoS) attack may lead to a worst case scenario.
- Availability: Every time a new packet arrives, data plane devices need a controller's involvement to process that packet. The overloaded controller may not be available for devices at all times. For example, NOX (Gude et al., 2008) can handle 30 K flow requests with a response time of less than 10 ms, but for larger networks, it may be higher (Tootoonchian et al., 2012; Karakus and Durresi, 2017).

Data Center Networks (DCNs), either copper (Pranata et al., 2019; Majidi et al., 2019) or fiber (Yang et al., 2015, 2016) based, are prime candidates where SDN is implemented, but due to its attractive features, enterprise level networks have also adopted SDN. The implementation of SDN at the enterprise level is difficult due to the issues discussed above. A simple solution is the distribution of SDN controllers.

Table 9
Summary of approaches used for virtualization.

Literature	Placement	Tenant's Controller	APIs Involved	SBI Support	Description
FlowVisor (Sherwood et al., 2010a)	Between Switch & Controller	M	SBI	OF	To provide an isolating between experimental traffic and data traffic
AutoSlice (Bozakov et al., 2012)	Between Switch & Controller	M	SBI	OF	Improve scalability by handling large number of flow tables
OpenVirteX (Al-Shabibi et al., 2014)	Between Switch & Controller	M	SBI	OF	Provides Topology, Address and Control Function Virtualization
FlowN (Drutskoy et al., 2013)	Inside Controller	S	NBI	OF	Allows tenants to run their own applications rather than having full control of network
Network Hypervisor (Huang and Griffioen, 2013)	Inside Controller	S	NBI	Multi Technology	Handles seamlessly different level of abstractions and a variety of APIs in SDN
NVP (Koponen et al., 2014)	Inside Controller	C	NBI	OVS	Allows cloud tenants to manage their data center resources
AutoVFlow (Yamanaka et al., 2014)	Between Switch & Controller	M	SBI	OF	Provides Complete "Flow Space" Virtualization in WAN
libNetVirt (Turull et al., 2012)	Inside Controller	S	NBI	Multi Technology	Provides a flexible way to create and manage virtual networks

Legend: M = Multiple, S=Single, C=Cluster, OF=OpenFlow, OVS=OpenVSwitch.

Such distributed controllers can be implemented as 1) Distributed (Flat) Architecture 2) Hierarchical Architecture. In a distributed architecture, all the controllers have equal rights and share information (e.g. topology, reachability, devices capabilities, etc.) with each other, as shown in Fig. 12(a). On the other hand, hierarchical architecture has two layers of controllers. The lower layer consists of domain controllers, sometimes referred to as the local controller, and the upper layer contains a root controller. Local controllers are responsible for their domain and update the root controller by using a control channel, as shown in Fig. 12(b). The root controller normally has more rights as compared to domain controllers and keeps network-wide information.

Table 10 presents a summary of these architectures along with the list of protocols used, their network types and languages used by different proposals. Some of the proposed architectures use distributed approaches, while others prefer hierarchical. Proposals such as FlowBroker and Orion use a mixture of these two approaches.

6.1.1. *Distributed architecture interfaces*

This architecture can be classified into two types. One is logically centralized but physically distributed and the other is completely distributed. In logically centralized but physically distributed SDN architecture, each controller is responsible for its domain but synchronized with other controllers. As soon as there is any change under any controller, it will update neighboring controllers. This enforces a consistent global view of the network. A key problem with this approach is that controllers consume network resources to provide information to each other and frequent change in the network may reduce network performance. On the other hand, in a completely distributed controller architecture, controllers are not synchronized. They may update each other using protocols, but consistency does become an issue in this approach, which may lead to unexpected behavior of the network.

SDNi (Yin et al., 2012) attempts to provide inter-controller communication in OpenDaylight which uses Akka and raft based synchronization protocol. It is a message exchange protocol among different domains coming under a single operator or collaborating operators. It exchanges customized messages like reachability, flow setup, and capability updates.

HyperFlow (Tootoonchian and Ganjali, 2010), an event-based solution, uses WheelFS (Stribling et al., 2009) as a file system for controller communication in a domain. It is a logically centralized and physically distributed architecture where switches connect to the nearest controller which updates neighboring controllers by using the publish/subscribe method. It provides a consistent global view of the network. HyperFlow runs as an application on top of NOX (Gude et al., 2008) controller and uses most of the features of NOX.

Onix (Koponen et al., 2010) is another approach to overcome scalability problems in SDN controllers, and provides flexibility for the development of applications in a distributed manner, by providing Distributed Hash Table (DHT) storage and group membership. It has three major components; (1) Network Information Base (NIB), (2) Partition and Cluster Aggregation for hierarchical structure, and (3) Consistency and Durability for applications. NIB is a data structure that maintains network entities. To access any particular entity, it queries the index of all entities by using the entity identifier. Moreover, NIB can cause scalability issues (e.g. exhaust system memory and saturate CPU or Onix instances) as it is not distributed. To resolve this issue, it uses partition and cluster aggregation. Control applications in Onix are used to partition the workload. Whereas in aggregation, cluster managed by different Onix nodes is considered as a single node. Moreover, authors claim that consistency and durability can be achieved by using different algorithms, however, details for these algorithms are missing.

Tam et al. (2011) used two approaches to resolve the problem of scalability and multipath among different controllers without using a global view in the data center environment. It uses multiple independent controllers to answer the request of underlying devices, instead of a single omniscient controller. The first approach is Path-Partition

Table 10
Summary for inter controller communication interfaces (EBIs).

Literature	Architecture	Protocol for Communication	Network Type	Prog. Language Used	Description
OpenDaylight (OpenDaylight, 2020; Yin et al., 2012)	Distributed	SDNi Using Akka and Raft	WAN	JAVA	SDN interface (SDNi) enables the controller to exchange information within the purview of define policies
Kandoo (Hassas Yeganeh et al., 2012)	Hierarchical	Messaging Channel	Data Center, Campus	C/C++/Python	Divides control plane in domain controller and root controller
DISCO (Phemius et al., 2014)	Distributed	AMQP	Data Center, Enterprise, WAN	JAVA	Based on Messenger and Agent Approach which are responsible for control information
Onix (Koponen et al., 2010)	Distributed	Zookeeper	Data Center, Enterprise	C++	Provides an API for the easiness in application development
HyperFlow (Tootoonchian and Ganjali, 2010)	Distributed	WheelFS	Data Center	C++	An event-based solution running on top of NOX and provide a consistent global view of the network
Tam et al. (Tam et al., 2011).	Distributed	Not Mentioned	Data Center	Controller Dependent	Allow controllers to distribute their loads to reduce response time in Data Center Environment
Elasticon (Dixit et al., 2014)	Distributed	Custom Protocol	Data Center, Cloud	JAVA	Ensures controller utilization by computing controller load and stretch or shrinks accordingly
ONOS (Berde et al., 2014)	Distributed	Raft	Enterprise, WAN	JAVA	Provides scalability and fault tolerance in control lane by using instance based approach
Helebrandt et al. (Helebrandt and Kotuliak, 2014)	Distributed	Custom Protocol	WAN	Not Mentioned	Enables Controller Communication using INT module responsible for connection establishment and keep alive messages
Yazici et al. (Yazici et al., 2014)	Distributed	JGroup	Data Center	JAVA	A Master controller is selected among different controllers by using Jgroup and a controller can be added or removed without network interruption
WE Bridge (Lin et al., 2014b)	Distributed	Custom BGP	Enterprise, WAN	JAVA	Provides Scalability and control messages are forwarded in JSON format
DMC (Chundrigar et al., 2016)	Distributed	RabbitMQ	Data Center, Enterprise, WAN	Python	Ensures flexibility in link and controller failure among heterogeneous domains
Bari et al. (Bari et al., 2013)	Distributed	Not Mentioned	WAN	Python	Solving Dynamic Controller Provisioning Problem and reduce flow setup time.
Orion (Fu et al., 2014)	Distributed & Hierarchical	Not Mentioned	WAN	JAVA	Solves path stretch problem and super linear complexity by combining distributed and hierarchal architecture
Bhole et al. (Bhole and Puri, 2015)	Hierarchical	Not Mentioned	WAN	JAVA	Improve communication reliability and reduce response time
FlowBroker (Marconett et al., 2015)	Distributed & Hierarchical	Broker Protocol	WAN	JAVA	Improves load balancing and network performance in multiple domains
Karakus et al. (Karakus and Durresi, 2015a)	Hierarchical	Broker Protocol	WAN	Controller Dependent	Providing Quality of Service using FlowBroker Approach
Guo et al. (Guo et al., 2015)	Hierarchical	NBI	WAN	JAVA	An hierarchical architecture using Northbound API for communication among different controllers
Wang et al. (Wang et al., 2016b)	Hierarchical	Restful API	Enterprise, WAN	Not Mentioned	Co-ordinate controller is used to provide communication among heterogeneous controllers
D-SDN (Santos et al., 2014)	Hierarchical	Custom Protocol	Home, WAN	Not Mentioned	Using Master and Secondary approach where master controller is delegating control to secondary

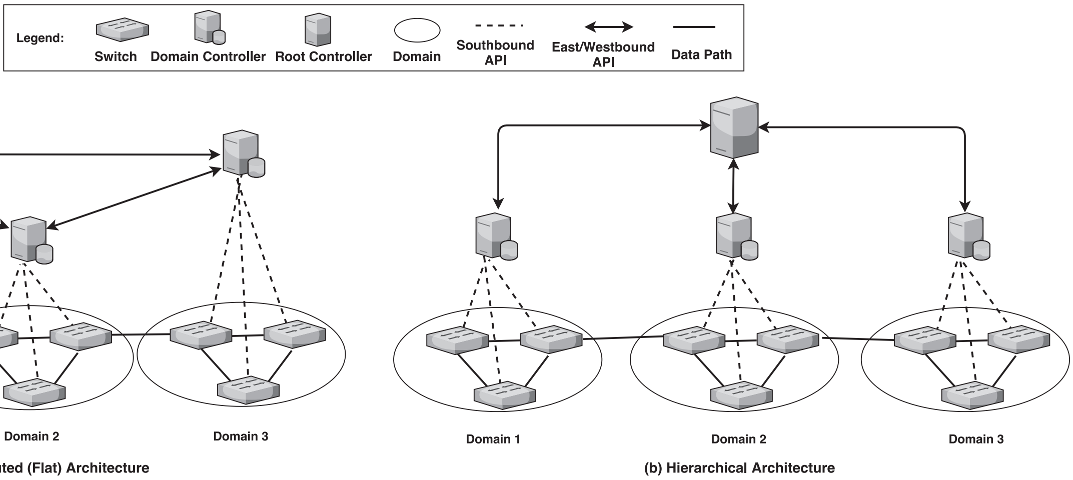


Fig. 12. Classification of distributed SDN architectures.

Approach; where all possible paths are calculated from source to destination using Dijkstra's algorithm and then each multipath is allocated to one of the controllers according to cost function and number of links monitored by that controller. Whereas, Partition-Path approach uses initially preferred links for path computation from source to destination. All the controllers have different routes of source and destination pairs. A source can send a request to all the controllers for the route. This solution will generate extra control traffic. Another solution is to have a mapping at source node where a table at source node describes which controller has a route for a particular destination.

Switch to controller mapping is static and a controller may become overloaded if a large number of flows arrive. It is quite possible that other controllers might be underutilized because of static mapping, which may lead to poor performance. Elasticon (Dixit et al., 2014) proposes an architecture, where a load of a controller is computed and the controller pool can be dynamically expanded and shrunk, which enhances the network performance and throughput. A switch can be connected to multiple controllers, with one master and rest as slaves, for fault tolerance purposes. A distributed data store is used to provide communication among different controllers and enables a logically centralized controller. It also has switch-specific information and each controller maintains a TCP connection with other controllers in the form of mesh. This TCP connection is used to send messages to coordinate with other controllers during switch migration. This connection is also used to send messages to a switch which is connected with other controllers as the master.

ONOS (Berde et al., 2014) also provides an approach for improving scalability and fault tolerance to the SDN control plane. It, first of all, creates a global network view by using Titan (Distributed graph database, 2020) graph database with Cassandra (Lakshman and Malik, 2010) key-value store for distribution. There are multiple instances but only one instance is the master for each switch, which is responsible for taking information from the switch and program it. One of the issues in the first ONOS prototype was the implementation of notifications and messaging across the ONOS instances. For changes in network states, ONOS modules had to check the database periodically which increases the CPU load as well as delay for reacting to events and communication among instances. This issue was resolved in the second prototype by creating an inter-instance communication module using a publish-subscribe mechanism based on Hazelcast (Hazelcast project, 2020).

To facilitate the deployment of SDN on large scale and to do traffic management by the coordination, Helebrandt et al. (Helebrandt and Kotuliak, 2014) proposed architecture for communication among multiple domains using an interface, which is referred as INT module. The protocol is divided into three sub-parts: 1) Controller Interconnection Session Control, 2) Capabilities Information Exchange, and 3) Path Setup. Controller interconnection session control is responsible for connection establishment. To reduce the administrative overhead, this session can be automated, but for security purposes, it has a manual setup. Capabilities Information Exchange is used to exchange the capabilities of network elements, whereas path setup is used for end to end flow setup. Another responsibility of this protocol is to send keepalive messages and provide updates to peer controllers. Furthermore, four types of messages are used for this purpose which are request, reply, help, and update. A sample packet header is also discussed for communication among multiple domains.

In (Yazici et al., 2014), Yazici et al. proposed a framework that provides support for dynamic addition and removal of controllers to the cluster without any interruption, and at the same time, the framework can work with numerous existing SDN controllers. It is a leader based approach where JGroup (Ban, 1998) notification and messaging infrastructure is used for the communication among different controllers to select the master controller. The master controller is responsible for delineation between different controllers and switches. If for any reason the master controller is not accessible, a new master is selected. This architecture is not hierarchical as it allocates only a few additional responsibilities to the master controller.

In DISCO (Phemius et al., 2014) authors used FloodLight OpenFlow Controllers in multiple domains. The solution provides Intra-Domain and Inter-Domain Communication and is resilient from disruptions. DISCO's architecture is mainly divided into two parts: Messenger and Agent. Messenger is responsible for lightweight control communication among different controllers by using Advanced Query Message Protocol (AMQP) (AMQP, 2020) using a publish-subscribe method, whereas, Agent supports network-wide functionalities like Connectivity, Monitoring, Reachability, and Reservation. Furthermore, agents identify alternative routes to offload traffic from weak interconnections. If they can not find any alternate routes, they reduce the frequency of control messages for these weak interconnections.

Implementation of WE Bridge (Lin et al., 2014b) uses heterogeneous controllers and provides a bridge for communication among these controllers. It first registers controllers and then provides virtualization among domains because it is possible that Internet domains may belong to different administrative authorities. Domains are using an interface to transfer messages in the JSON format whereas other options are also recommended like; XML and Yanc. Two different applications Inter-Domain Path Computation and Source-Address based Multipath Routing run on top of controllers.

Distributed Multi-Domain Controller (DMC) (Chundrigar et al., 2016) connects heterogeneous networks. This provides the privacy of domains and at the same time deals with link and controller failure among different domains. Light-weight controller-to-controller communication is done by using RabbitMQ (RabbitMQ, 2020) which is an implementation of AMQP. The controller in each domain is event-driven and provides services in its domain and communicates with neighboring domains by using a control channel that is integrated with the REST interface. A centralized database is managed where all the controllers update data. The Publish-Subscribe method is used among controllers.

6.1.2. Hierarchical architecture interfaces

In hierarchically distributed architecture there are two layers of SDN controllers. Lower layer controllers are responsible for their respective domains. At the upper layer, the root controller is responsible for managing a group of domain controllers. Control information in this architecture is less as compared to flat architecture but Single Node Failure problem still exists, however it is not as problematic as earlier.

To make SDN scalable, several frequent events (e.g. network-wide statistics collection and flow arrivals) in the control plane must be reduced. It can be done by processing these events in a data plane which is a costly solution, as it requires switch modifications. A solution to this problem is addressed by Kandoo (Hassas Yeganeh et al., 2012) which is a hierarchical control plane architecture with two layers of controllers. The lower layer of controller deals with their domains and does not have a network-wide view. Moreover, controllers in this layer are subjected to deal with frequent events. Whereas, the top layer which consists of a root controller and has the global network state and processes rare events (e.g. elephant flows). It uses an API in terms of two applications as App_{detect} and $App_{reroute}$. App_{detect} runs on local controllers and constantly queries each switch to detect elephant flows. $App_{reroute}$ runs on root controller and install flow entries on switches if elephant flow is detected by App_{detect}. To differentiate the applications running on a local or root controller, a flag is used.

Karakus et al. (Karakus and Durresi, 2015b) also proposed architecture with two levels which can be extended. The bottom level is referred to as network level and contains different SDN domains handled by local controllers. Broker level is an upper level where a super controller is placed which supervises domain controllers. Local controllers advertise all their reachable addresses as well as border switches connecting their neighboring domains by using the eastbound interface, which is file-based, to the super controller. It allows the super controller or broker to determine the source and destination domains. Whenever there is inter-domain communication, the broker asks source and destination domains to advertise all QoS paths from source to gateways (in case of source domain) and destination to gateways (in case of destination domain). As the broker has the global view of the network, it determines all paths from source to destination. After computing the best route, the broker sends ingress and egress node points to respective domains (i.e. source, destination, and transit) to reserve the QoS values. Authors also claim that a controller in a hierarchical setting handles 50% less traffic than a controller in a non-hierarchic environment.

Guo et al. (2015) used a hierarchical model for multi-domain controller communication, where local controllers are responsible for their domains whereas the coordinating controller is responsible for the global view of the network and provides inter-controller communication which is prototyped in Java. Communication among coordinate controller and domain controller as well as with the applications is done by using Northbound API. This NBI can provide information about local controller to applications and coordinating controller. It also enables applications and Coordinate controller to configure flow tables and traffic forwarding. Furthermore, two modules are implemented as Topology Management and Flow Management. Topology Management is responsible to get the whole topology and flow management is about updating and installing flows to domain controllers and data plane respectively.

To solve the issue of consistency among diversified controllers, Wang et al. (2016b) proposed a coordinate controller approach which helps for communication among heterogeneous controllers. The control plane in this architecture is divided into two parts, coordinate controller and domain controller. The coordinate controller is responsible for the collection of information of the whole network and domain controllers and dynamic controllers may use different technologies. The domain controller is a traditional SDN controller and running its domain. Protocol interpreter is used for eastbound communication among coordinate controller and domain controller which enables the coordinate controller to implement end-to-end provisioning services across multiple domains. To resolve the issue of diversity of vendors, it uses a unified Northbound interface. This unified API is divided into two parts: the topology API and service API. Topology API is used for the collection of network information and elements connectivity to design a global view of the network. The purpose of the service API is to launch service requests and set up a connection in the network.

Sometimes control can also be delegated to underlying devices. For example, Decentralized SDN (D-SDN) (Santos et al., 2014) is a hybrid approach using Main Controller (MC) and Secondary Controller (SC) by delegating control. It allows physical as well as logical control distribution by using MC and SC. One of the integral features in D-SDN is security, as MC authorizes before delegating control to SC so that it can act as a controller. This delegation occurs upon a request from SC that is triggered by a set of events. These events include a newly installed SC or a gateway through which mobile devices can access the Internet. SC cannot write any new flow entries without authentication of MC. Communication between MC and SC is done by using an interface for control delegation message where SC requests a Check_in_Request and whereas master authorizes or denies Check_in_Response. Similarly, communication among SCs is done by using D-SDN's SC-SC protocol to implement fault tolerance. Switch to controller mapping in SCs are master-slave based. A slave SC can receive Hello messages for a predefined time. If it does not receive, slave SC can become master by sending role_change message to the switch.

Every controller has different features and northbound interfaces, for example, deleting a flow in Floodlight controller is easier as compared to POX and both of these controllers have different functions. Zebra (Yu et al., 2015) attempts to resolve this heterogeneity problem by dividing the control layer into two parts: the dissemination layer and the decision layer. Dissemination Layer has traditional SDN controllers, whereas, two main modules referred to as Heterogeneous Controller Management (HCM) and Domain Relationships Management (DRM) are placed in the decision layer. Different SDN controllers (e.g. Floodlight, POX, OpenDaylight, etc.) are placed in HCM and it handles the routing decision inside a domain. Whereas, CRM provides decision making among multiple domains.

6.1.3. Hybrid architecture interfaces

Orion (Fu et al., 2014) is an example of large scale networks with a mixture of distributed (flat) and hierarchical architectures. It mainly focuses on the Path Stretch problem and Super Linear Computation Complexity problems introduced by these two architectures. Path stretch is the difference between the best optimal path and actual path traffic takes in the network. This problem occurs in hierarchical architecture. A superlinear computational complexity issue exists in distributed (flat) architecture and normally occurs because of the size of the network. Orion addresses these issues by dividing architecture into three layers which include, physical layer, area controller layer, and domain controller layer. Area controllers are close to OpenFlow switches and pass on the information to domain controllers which consider area controllers as nodes and reduce the path stretch and super linear computational complexity problem. A TCP connection is used as an interface for communication purposes between the area controller and the domain controller. This channel is used for sending requests and distribute rules.

In (Marconett et al., 2015), Marconett et al. proposed a mechanism called FlowBroker using a hierarchical approach to do load balancing and improve network performance. Brokers work as root controller on the top layer, whereas, in the lower layer, domain controllers are managing their domains. Each domain controller is attached to a broker according to their reputation in terms of load balancing and performance. This performance reputation by using machine learning based agents that are connected with domain controllers. A broker is a software process that allows the exchange of network-wide state along with the flow table updates to respective domain controllers. To counter the failover mechanism, switch controller mapping is done by using primary and secondary controllers. Secondary controllers monitor the primary controller by using an interface based on Ctrl_Keep_Alive messages every 2 s. Whereas, the primary controller sends Ctrl_Table_Backup messages periodically to mirror any changes that occur in the primary controller.

6.1.4. Security in inter-controller communication

In SDN, clustering provides many advantages which include scale-out performance, high availability, and data durability. However, it also brings a few new challenges. Similar to SBI, inter-controller communication in SDN is not secure by default which allows hackers to observe and modify traffic (Secci et al., 2017). ONOS (Berde et al., 2014) created a performance and security brigade in early 2017 to stress various components of ONOS in terms of security and performance. To make this communication secure, this brigade proposed that TLS can be used which ensures secure communication in cluster (Secci et al., 2018).

6.1.5. Insights on SDN domain interaction

Distributed controller approaches solve several issues in SDN, but it also introduces some new challenges, for example, consistency and resource utilization. Similar to the northbound interface, east interfaces are also not standardized which is one of the major challenges in distributed controllers. Moreover, a consistent global view is also required in distributed controllers which may reduce network performance. Controlling the overhead is another major challenge. From a wireless network perspective, lightweight controllers for access points could lead to better flow management in mobile networks. Hence, EBIs for inter-AP Controller communication will certainly benefit the softwarization of wireless networks.

6.2. Interaction between SDN and traditional networks (westbound APIs)

Some critics believe that Inter-Domain communication in the legacy network is better rather than using SDN. For example, in (Wibowo and Gregory, 2016) authors use BGP on top of TCP for inter-controller communication. Session starts by using *open* message which leads to *established* once the connection is established among these controllers. Controllers can share information by using *update* messages which include reachability and messages like bandwidth information. Authors claim that legacy networks are performing well as compared to SDN with BGP and SDN without BGP.

SDN is a promising way to re-architect the Internet and transitioning from traditional network to SDN is an important issue as there are a large number of Autonomous Systems throughout the globe (Yangyang and Jun 2020; Gupta et al., 2014). During this transition SDN must co-exist with legacy network and any SDN network should be able to share reachability information, forward traffic, and express routing policies with a traditional network through gateways in the SDN domain. Fig. 13 presents traditional ASes communication with the SDN domain through border gateway switches. Migration Work Group (Migration working group, 2020) under ONF (ONF, 2020) previously worked on proposals for the transition from traditional to SDN networks, however, it is not functional anymore.

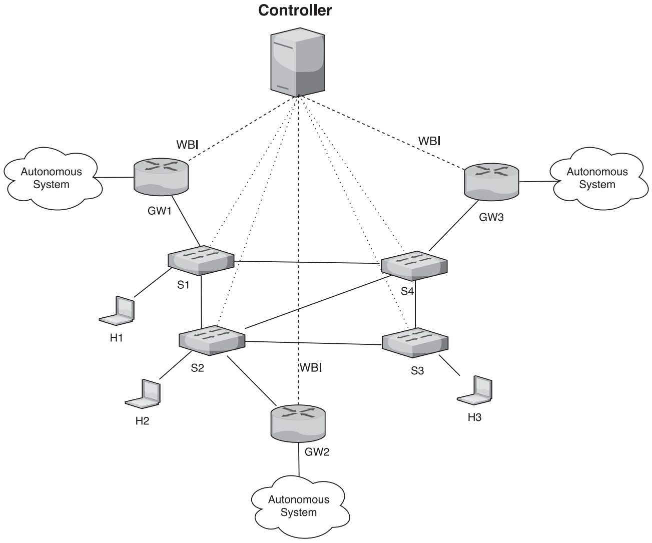


Fig. 13. SDN and autonomous systems.

RouteFlow (Nascimento et al., 2011) is one of the first approaches of IP routing on OpenFlow switches which is composed of RouteFlow Server, RouteFlow Slave, and RouteFlow Controller. RouteFlow Controller runs as an application on top of the SDN Controller. RouteFlow Server keeps network-wide state and core logic resides in it and manages a virtual network environment to interconnect virtualized IP routing engines e.g. Quagga (Quagga routing suite, 2020). For legacy network, RouteFlow Slave is used which updates the server using a custom interface (i.e. RouteFlow Protocol). The messages of this protocol are either command type or event type. These messages are the subset of OpenFlow protocol along with some other messages (e.g. send updates, accept or reject VM, RF-Slave configuration, etc.).

Another solution using BGP is SDN-IP (Lin et al., 2013) where the seamless interconnection between the SDN domain and the traditional domain is focused. SDN-IP Peering application (having BGP Route Module and Proactive Flow Installer Module) runs on top of the network operating system. BGP route module synchronizes BGP route updates pushed by the BGP process which is ZebOS BGPd (ZebOS, 2020) and stores them in Route Information Base (RIB) which can scale to 10,000 entries, whereas Proactive Flow Installer uses routes learned through BGP and installs flow entries accordingly. In simple words, the controller of the SDN domain uses BGP to exchange routing information with neighboring legacy network domains but uses SDN's centralized mode to control local AS's BGP route calculation and installation.

BTSDN (Lin et al., 2014a) proposes a practical solution by integrating SDN network to the current Internet with BGP (Rekhter and Hares, 2020; Claffy, 2012). Using BTSDN, SDN and traditional networks can co-exist by using Internal BGP (iBGP) and external BGP (eBGP) so that SDN can be incrementally deployed to the Internet and finally replace traditional networks. Usage of eBGP and iBGP in BTSDN is the same as a traditional network. OpenFlow switches directly connected to border routers and play a key role and act as a proxy because the SDN controller cannot directly control border routers. However, the controller can install certain flow entries on these SDN switches. The data plane adopts the mechanism of Address Resolution Protocol (ARP) and Media Access Control (MAC) to ensure the delivery of IP packets between SDN and traditional domains.

Internet eXchange Points (IXPs) are playing an integral role in interconnecting many networks and bringing popular content closer to end-users. Routing in a traditional network using BGP only utilizes destination IP prefix, hence it cannot make fine-grained decisions based on the type of application or sender. Similarly, routes are learned from direct neighbors so the network cannot provide proper end to end service. At the same time network can not express inbound and outbound paths. To resolve all these issues a combination of IXPs and SDN makes Software Defined eXchange (SDX) (Gupta et al., 2014). It enables their participants to run novel applications, written in Pyretic (Monsanto et al., 2013), that control the flow of traffic entering and leaving their border routers. It gives an illusion to each AS, that a virtual SDN switch is connected to its border router and enables flexible specifications of forwarding policies. It also provides isolation among different participants.

Insights: Newer versions of OpenFlow provides hybrid solutions, where controllers can communicate with SDN elements as well as traditional switches. However, research for the westbound interface is required during the transition period from traditional networks to SDN. Co-existence interoperability will be a key step for large scale SDNs. Moreover, it will be important to address new technologies and architectures such as data-centric networks. Some recent research efforts have focused on SDN in data-centric communication, however westbound APIs between pure-SDN & data-centric-SDN domains will certainly be challenging.

7. Future research directions

Software Defined Networks have been a major research area in recent years, however, more effort has been placed on controller design. Interfaces, on the other hand, has received less attention, except OpenFlow. Here we present several research directions and possible challenges for each type of interface.

7.1. Southbound interface

OpenFlow has dominated the SBI, as it has matured rapidly, although other solutions (as discussed in this article) also offer interesting features. OpenFlow has undergone rapid evolution, which has its pros and cons. A long header for matching is used in different OpenFlow versions, which leads to the requirement of more storage for rules and takes more processing time. Although several other studies exist which address this size issue, not all have been integrated into OpenFlow. This can certainly be a development direction. An optimal mechanism to reduce the storage of processing requirements will be beneficial to the overall system in different domains.

ForCES offers a rich set of features, such as separation of control and data plane without changing the architecture of traditional networks, and extensibility. These features are still not available in OpenFlow or other southbound interfaces. A possible approach could be to further develop ForCES to become a more elaborate solution, or to manage these features in OpenFlow. Combining all possible features in a single solution may make it too complex and increase its overhead, hence further research is needed to adequately evaluate the performance of both, and then develop on their capabilities for specific types of networks.

Two different modes are used to reduce the latency in the flow rule setup: proactive and reactive. In proactive mode, flows are already installed before the arrival of packets. However, this solution creates unnecessary overhead but can be useful for critical flows which are delay sensitive. In reactive mode time taken for flow, installation is crucial because flow installation is done after packet arrival. This mode is not suitable for time-sensitive flows. Research challenges in this regard are multifold, and depending on the type of network the solutions could be different. An SBI that takes into account the design and usage of networks, types of communicating devices and their mobility traffic patterns may yield better flow installation time. The literature is also missing any performance analysis in terms of SBIs effect on flow installation by different solutions. This is another area of exploration.

For a wide deployment of SDN in WSNs, some issues have already been addressed. However, robustness in the case of sink failure is a challenge and an open question. As more sensor node deployment is being done in different networks, hence it is important to evaluate the softwarization and thus a unified lightweight SBI. The memory size of sensor nodes is limited to keep a large number of flow rules. Similarly, the security and mobility management of different nodes requires further research attention. Similarly, research in the area of energy management is a necessity to ensure efficient use of resources for communication among different SDN planes and elements.

A large number of devices are expected to be connected through the Internet because of the power of the Internet of Things. SDN can help IoT in different aspects. However, to manage the enormous collection of heterogeneous devices via centralized control, an adequate solution in terms of the southbound interface is still missing. The fundamental limitation is in the OpenFlow design context. Current solutions are only designed to communicate with switches. However, in a multi-hoping ad-hoc nature of IoT, these solutions have to effectively extend beyond vSwitch. This again requires a newer and efficient design for SBI which can communicate with heterogeneous devices with multi-technology interfaces. A unified multi-technology flow installation, topology management, and configuration SBI will be required in the future for large scale IoT systems.

7.2. Northbound interface

SDN provides incredible opportunities for network operators in terms of network management using a centralized controller. However, due to the absence of a standardized API, this has become a challenging task. Moreover, it becomes more challenging and time-consuming due to distributed controller environments. Hence, to make SDN a powerful option for network operators, an instinctive API is desired. Due to the diverse landscape of network elements and multiple versions of protocols (e.g. OpenFlow) portable application development is very difficult. Thus, a flexible interface is required which is capable of removing this underlying complexity.

Several new features have been introduced in all OpenFlow versions, and programming languages can take advantage of these features. However, FatTire (Reitblatt et al., 2013) is the only language for NBI which incorporated group tables introduced in OpenFlow version 1.1. Similarly, there are many other features of OpenFlow which can help in various programming languages, but very limited advantages have been taken by high-level languages. Efforts in this regard can lead to a potential increase in application development for SDN in different domains. Many of the application programming languages offer libraries and community contributed extensions which make them famous in developer communities. However, SDN programming languages do not provide such an interface or repository. Instead, these languages provide fundamental constructs that force developers to write applications from scratch.

Vendor-specific and ad-hoc solutions are major issues of traditional networking which exist in SDN as well as controller-based northbound interfaces. Intent-based interfaces can be a solution to this issue, however, further exploration of how to utilize them effectively is required.

7.3. Virtualization

In SDN, data plane performance is directly dependent on control plane performance. There is significant research to enhance the performance of the control plane. For example, switches are statically assigned to controllers which may increase response time. DCAP (Wang et al., 2017b) addressed this issue by dynamically assigning controllers to minimize response time. However, different tenants of vSDN may also need to specify their control plane demands in addition to requesting the virtual topologies and links. Similarly, the tenant may specify the demands for OpenFlow message types. For example, a tenant may require fast processing of FLOW_MOD messages instead of OpenFlow stats. In the current research, defining these control planes and OpenFlow specific demands is not available. This research direction, in the long run, will allow fine-grained virtualization and API configuration.

Another important issue is reliability and fault tolerance of hypervisors. Different mechanisms and procedures need to be defined to recover faults and failures of hypervisors. A single hypervisor may not be sufficient to manage a large number of vSDN elements. To overcome the scalability issue, the distributed architecture of hypervisors can also be an interesting research area. Similar to controller placement, hypervisor placement plays a significant role in the overall system. Research in optimizing this placement is yet another challenge.

7.4. East/westbound interface

Just like the northbound interface of SDN, communication between different controllers is not established by a universally accepted protocol. The communication interface between multiple controllers directly affects performance. Thus, a standardized API and protocol are required so network performance can be enhanced. Due to this reason, SDN deployment is difficult in large scale networks. Also, SDN controllers play a critical role in managing and monitoring network traffic. However, multi-controller architecture still lacks in safety mechanisms as well as suspicious traffic detection.

The different types of controllers and their architectures also introduces heterogeneity challenges. As many of the controllers have matured over time, providing a unified eastbound interface is a challenging task. Similarly, ensuring the inter-controller communication overhead to be as minimal as possible, first requires evaluation and comparison of existing solutions, and then requires the development of an efficient communication interface. These challenges are not isolated, but also affect the performance of other interfaces, such as SBI which have to install flows provided by root/master controller.

Westbound interface with traditional networks may need software implementation on routers. Such implementations may translate the flow entries to routing paths, and vice versa. Similarly, interaction with the non-IP domain (such as data-centric networks) is another major research direction, where the translation of IP flows to name-based flows will be interesting. OF will also need to adapt in this regard.

8. Conclusion

This paper presented a detailed and systematic survey of different types of interfaces and interface protocols for Software Defined Networks. These protocols are necessary for inter-layer and inter-element communication in the complete SDN architecture. We classify these interfaces based on their directional-communication properties, and then further sub-classify them based on the functionality. Southbound interfaces between the control plane and data plane have been dominated by OpenFlow. It has become a de facto industry standard, although some other SBI solutions are also available which are not dependent on OpenFlow. Most of the research work done in SBIs is an extension or improvement of the OpenFlow protocol. However, it has limited applications for emerging technologies such as IoT. Northbound interfaces between application plane and controllers are quite different from SBIs as the purpose is to enable users to control, configure, and program the network. Hence, they are classified in terms of programmability, portability, controller-based, and intent-based solutions. Although virtualization is highly integrated into SDN, we present it as a separate functional element and review the different interfaces which interact with hypervisors. Inter–SDN domain interfaces and SDN to traditional network interfaces have also been analyzed in detail for their different architectures and properties. In addition to the insights and future directions presented earlier, it is important to note that network programmability will have a major impact on next-generation Internet architecture, either it is wired or wireless. Thus, with the emergence of new technologies, the interface protocols will have to evolve at the same pace as the controllers and other technologies.

Acknowledgment

This work is in part supported by the National Natural Science Foundation of China No. 61772077, 61370192, and the Beijing Natural Science Foundation No. 4192051.

References

Agborubere, B., Sanchez-Velazquez, E., June 2017. Openflow communications and tls security in software-defined networks. In: 2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber. Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pp. 560–566.

Al-Shabibi, A., Leenheer, M.D., Gerola, M., Koshibe, A., Snow, W., Parulkar, G., 2014. Openvirtex: a network hypervisor. In: Open Networking Summit 2014 (ONS 2014). Plus 0.5em Minus 0.4emSanta Clara. USENIX Association, CA.

Ali, N.F., Said, A.M., Nisar, K., Aziz, I.A., Nov 2017. A survey on software defined network approaches for achieving energy efficiency in wireless sensor network. In: 2017 IEEE Conference on Wireless Sensors. ICWiSe, pp. 1–6.

AMQP, 2020. Advance message queuing protocol. Accessed on: 01-Jan-2020. [Online]. Available: https://www.amqp.org.

Anadiotis, A.C.G., Milardo, S., Morabito, G., Palazzo, S., 2018. Towards unified control of networks of switches and sensors through a network operating system. IEEE Intern. Things J. 99 11.

Anderson, C.J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., Walker, D., Jan. 2014. Netkat: semantic foundations for networks. SIGPLAN Not 49 (1), 113–126.

Badotra, S., Panda, S.N., 2019. Evaluation and comparison of opendaylight and open networking operating system in software-defined networking. Clust. Comp..

Ban, B., 01 1998. Design and Implementation of a Reliable Group Communication Toolkit for Java.

Banks E., Thinking about SDN? Here are 42 vendors that offer SDN products, Accessed on: 01-Jan-2020. [Online]. Available: https://searchsdn.techtarget.com/news/2240212374/Thinking-about-SDN-Here-are-42-vendors-that-offer-SDN-products.

Banse, C., Rangarajan, S., Aug 2015. A secure northbound interface for sdn applications. In: 2015 IEEE Trustcom/BigDataSE/ISPA, vol. 1, pp. 834–839.

Bari, M.F., Roy, A.R., Chowdhury, S.R., Zhang, Q., Zhani, M.F., Ahmed, R., Boutaba, R., Oct 2013. Dynamic controller provisioning in software defined networks. In: Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013), pp. 18–25.

Belter, B., Binczewski, A., Dombek, K., Juszczyk, A., Ogrodowczyk, L., Parniewicz, D., Stroiski, M., Olszewski, I., Sept 2014. Programmable abstraction of datapath. In: 2014 Third European Workshop on Software Defined Networks, pp. 7–12.

Bera, S., Misra, S., Vasilakos, A.V., Dec 2017. Software-defined networking for internet of things: a survey. IEEE Intern. Things J. 4 (6), 1994–2008.

Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., Parulkar, G., 2014. Onos: towards an open, distributed sdn os. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, Ser. HotSDN 14. Plus 0.5em Minus 0.4em. ACM, New York, NY, USA, pp. 1–6.

Berman, M., Chase, J.S., Landweber, L., Nakao, A., Ott, M., Raychaudhuri, D., Ricci, R., Seskar, I., Geni, Mar. 2014. A federated testbed for innovative network experiments. Comput. Network. 61, 5–23.

Bhole, P.D., Puri, D.D., Dec 2015. Distributed hierarchical control plane of software defined networking. In: 2015 International Conference on Computational Intelligence and Communication Networks (CICN), pp. 516–522.

Bianchi, G., Bonola, M., Capone, A., Cascone, C., Apr. 2014. Openstate: programming platform-independent stateful openflow applications inside the switch, *SIGCOMM Comput*. Commun. Rev. 44 (2), 44–51.

Bizanis, N., Kuipers, F.A., 2016. SDN and virtualization solutions for the internet of things: a survey. IEEE Access 4, 5591–5606.

Blenk, A., Basta, A., Reisslein, M., Kellerer, W., 2016. Survey on network virtualization hypervisors for software defined networking. IEEE Commun. Surv. Tutor. 18 (1), 655–685 Firstquarter.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D., Jul. 2014. P4: programming protocol-independent packet processors, *SIGCOMM Comput*. Commun. Rev. 44 (3), 87–95.

Boulis, A., Han, C.-C., Shea, R., Srivastava, M.B., Sensorware, Aug. 2007. Programming sensor networks beyond code update and querying. Pervasive Mob. Comput. 3 (4), 386–412.

Bozakov, Z., Papadimitriou, P., Autoslice, 2012. Automated and scalable slicing for software-defined networks. In: Proceedings of the 2012 ACM Conference on CoNEXT Student Workshop, Ser. CoNEXT Student 12. Plus 0.5em Minus 0.4em. ACM, New York, NY, USA, pp. 3–4.

Bradley, J.M., 2013, Nov. The internet of everything: creating better experiences in unimaginable ways. [Online]. Available: https://blogs.cisco.com/digital/the-internet-of-everything-creating-better-experiences-in-unimaginable-ways.

Campbell, A.T., De Meer, H.G., Kounavis, M.E., Miki, K., Vicente, J.B., Villela, D., Apr. 1999. A survey of programmable networks, *SIGCOMM Comput*. Commun. Rev. 29 (2), 7–23.

Casey, C.J., Sutton, A., Sprintson, A., 2014. tinynbi: distilling an API from essential openflow abstractions. CoRR abs/1403, 6644.

Chen, X., Wu, T., Jan 2017. Towards the semantic web based northbound interface for sdn resource management. In: 2017 IEEE 11th International Conference on Semantic Computing. ICSC, pp. 40–47.

Ching-Hao, C., Lin, Y.-D., 2015. Openflow version roadmap. Accessed on: 01-Jan-2020. [Online]. Available: http://speed.cis.nctu.edu.tw/ydlin/miscpub/indep_frank.pdf.

Chundrigar, S.B., Shieh, M.-Z., Tung, L.-P., Lin, B.-S.P., 2016. Dmc: Distributed Approach in Multi-Domain Controllers. in *INC*, pp. 31–36.

Ciciolu, M., alhan, A., Hubsflow, 2019. A novel interface protocol for sdn-enabled wbans. Comput. Network. 160, 105–117.

Claffy, K., 2012. Border gateway protocol (bgp) and traceroute data workshop report. SIGCOMM Comput. Commun. Rev. 42 (3), 28–31.

Conti, M., Kaliyar, P., Lal, C., 2019. Censor: cloud-enabled secure iot architecture over sdn paradigm. Concurrency Comput. Pract. Ex. vol. 0, no. 0, p. e4978, e4978 cpe.4978.

Costanzo, S., Galluccio, L., Morabito, G., Palazzo, S., Oct 2012. Software defined wireless networks: unbridling sdns. In: 2012 European Workshop on Software Defined Networking, pp. 1–6.

Cox, J.H., Chung, J., Donovan, S., Ivey, J., Clark, R.J., Riley, G., Owen, H.L., 2017. Advancing software-defined networks: a survey. IEEE Access 5, 25487–25526.

Curtis, A.R., Mogul, J.C., Tourrilhes, J., Yalagandula, P., Sharma, P., Banerjee, S., Devoflow, Aug. 2011. Scaling flow management for high-performance networks, *SIGCOMM Comput*. Commun. Rev. 41 (4), 254–265.

Dasu, T., Kanza, Y., Srivastava, D., 2017. Geotagging ip packets for location-aware software-defined networking in the presence of virtual network functions. In: Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Ser. SIGSPATIAL 17. Plus 0.5em Minus 0, vol. 9. ACM, 4emNew York, NY, USA. 19:4.

Desai, A., Nagegowda, K.S., Ninikrishna, T., March 2016. A framework for integrating iot and sdn using proposed of-enabled management device. In: 2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT), pp. 1–4.

Dhawan, M., Poddar, R., Mahajan, K., Mann, V., 2015. Sphinx: Detecting Security Attacks in Software-Defined Networks, 01.

Distributed graph database, Accessed on: 01-Jan-2020. [Online]. Available: http://titan.thinkaurelius.com/.

Dixit, A., Hao, F., Mukherjee, S., Lakshman, T.V., Kompella, R.R., Oct 2014. Elasticon; an elastic distributed sdn controller. In: 2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems. ANCS), pp. 17–27.

DLUX, 2020. OpenDaylight DLUX: DLUX Karaf feature. Accessed on: 01-Jan-2020. [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight_DLUX:DLUX_Karaf_Feature.

Drutskoy, D., Keller, E., Rexford, J., March 2013. Scalable network virtualization in software-defined networks. IEEE Internet Comput. 17 (2), 20–27.

Dvir, A., Haddad, Y., Zilberman, A., 2019. The controller placement problem for wireless sdn. Wireless Network 25 (8), 4963–4978.

Enns, R., Bjorklund, M., Schoenwaelder, J., Bierman, A., 2011. Network configuration protocol (netconf). Accessed on: 01-Jan-2020. [Online]. Available: https://www.rfc-editor.org/rfc/pdfrfc/rfc6241.txt.pdf.

Fei, X., Liu, F., Xu, H., Jin, H., June 2017. Towards load-balanced vnf assignment in geo-distributed nfv infrastructure. In: 2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS), pp. 1–10.

Fei, X., Liu, F., Xu, H., Jin, H., 2018. Adaptive vnf scaling and flow routing with proactive demand prediction. In: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, 4, pp. 486–494.

Ferguson, A.D., Guha, A., Liang, C., Fonseca, R., Krishnamurthi, S., Aug. 2013. Participatory networking: an api for application control of sdns. SIGCOMM Comput. Commun. Rev. 43 (4), 327–338.

Fonseca, P.C.d.R., Mota, E.S., 2017. A survey on fault management in software-defined networks. IEEE Commun. Surv. Tutor. 19 (4), 2284–2321 Fourthquarter.

Foster, N., Harrison, R., Freedman, M.J., Monsanto, C., Rexford, J., Story, A., Walker, D., Sep. 2011. Frenetic: a network programming language. SIGPLAN Not. 46 (9), 279–291.

Fu, Y., Bi, J., Gao, K., Chen, Z., Wu, J., Hao, B., Oct 2014. Orion: a hybrid hierarchical control plane of software-defined networking for large-scale networks. In: 2014 IEEE 22nd International Conference on Network Protocols, pp. 569–576.

Galluccio, L., Milardo, S., Morabito, G., Palazzo, S., April 2015. Sdn-wise: design, prototyping and experimentation of a stateful sdn solution for wireless sensor networks,. In: 2015 IEEE Conference on Computer Communications (INFOCOM), pp. 513–521.

gRPC, 2020. General purpose remote procedure call. Accessed on: 01-Jan-2020. [Online]. Available: https://grpc.io/docs/guides/.

Gong, Y., Huang, W., Wang, W., Lei, Y., Dec 2015. A survey on software defined networking and its applications. Front. Comput. Sci. 9 (6), 827–845.

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

Guo, Z., Hu, Y., Shou, G., Guo, Z., Sept 2015. An implementation of multi-domain software defined networking. In: 11th International Conference on Wireless Communications, Networking and Mobile Computing. WiCOM 2015, pp. 1–5.

Gupta, A., Vanbever, L., Shahbaz, M., Donovan, S.P., Schlinker, B., Feamster, N., Rexford, J., Shenker, S., Clark, R., Katz-Bassett, E., 2014. Sdx: a software defined internet exchange. In: Proceedings of the 2014 ACM Conference on SIGCOMM, Ser. SIGCOMM 14. Plus 0.5em Minus 0.4em. ACM, New York, NY, USA, pp. 551–562.

Haleplidis, E., Salim, J.H., Halpern, J.M., Hares, S., Pentikousis, K., Ogawa, K., Wang, W., Denazis, S., Koufopavlou, O., 2015. Network programmability with forces. IEEE Commun. Surv. Tutor. 17 (3), 1423–1440.

Hardt, D., 2012. The OAuth 2.0 authorization framework. Accessed on: 01-Jan-2020. [Online]. Available: https://tools.ietf.org/html/rfc6749.

S.Hares, Analysis of comparisons between OpenFlow and ForCES, Accessed on: 01-Jan-2020. [Online]. Available: https://tools.ietf.org/pdf/draft-hares-forces-vs-openflow-00.pdf.

Hassas Yeganeh, S., Ganjali, Y., Kandoo, 2012. A framework for efficient and scalable offloading of control applications. In: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, Ser. HotSDN 12. Plus 0.5em Minus 0.4em. ACM, New York, NY, USA, pp. 19–24.

Hazelcast project, Accessed on: 01-Jan-2020. [Online]. Available: https://hazelcast.org/.

Helebrandt, P., Kotuliak, I., Dec 2014. Novel sdn multi-domain architecture. In: Emerging eLearning Technologies and Applications (ICETA), 2014 IEEE 12th International Conference on, pp. 139–143.

Hernan, S., Lambert, S., Ostwald, T., Shostack, A., 2006. Uncover security design flaws using the stride approach. Accessed on: 01-Jan-2020. [Online]. Available: https://adam.shostack.org/uncover.html.

Hinrichs, T.L., Gude, N.S., Casado, M., Mitchell, J.C., Shenker, S., 2009. Practical declarative network management. In: Proceedings of the 1st ACM Workshop on Research on Enterprise Networking, Ser. WREN 09. Plus 0.5em Minus 0.4em. ACM, New York, NY, USA, pp. 1–10.

Hu, T., Guo, Z., Baker, T., Lan, J., 2018. Multi-controller Based Software-Defined Networking: A Survey, vol. 99. *IEEE Access*. 11.

Huang, S., Griffioen, J., July 2013. Network hypervisors: managing the emerging sdn chaos. In: 2013 22nd International Conference on Computer Communication and Networks (ICCCN), pp. 1–7.

Intent framework, Accessed on: 01-Jan-2020. [Online]. Available: https://wiki.onosproject.org/display/ONOS/IntentFramework.

Internet engineering task force, Accessed on: 01-Jan-2020. [Online]. Available: https://www.ietf.org/.

Karakus, M., Durresi, A., March 2015. A scalable inter-as qos routing architecture in software defined network (sdn). In: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, pp. 148–154.

Karakus, M., Durresi, A., March 2015. A scalable inter-as qos routing architecture in software defined network (sdn). In: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, pp. 148–154.

Karakus, M., Durresi, A., 2017. A survey: control plane scalability issues and approaches in software-defined networking (sdn). Comput. Network. 112, 279–293 Supplement C.

Karaf, 2020. Apache Karaf Applications Runtime. Accessed on: 01-Jan-2020. [Online]. Available: https://karaf.apache.org/.

N. P. Katta, J. Rexford, and D. Walker, Logic programming for software-defined networks, Accessed on: 01-Jan-2020. [Online]. Available: http://frenetic-lang.org/publications/logic-programming-xldi12.pdf.

Kemp, S., 2018. Digital in 2018: world's internet users pass the 4 billion mark. Accessed on: 01-Jan-2020. [Online]. Available: https://wearesocial.com/blog/2018/01/global-digital-report-2018.

Khan, I., Belqasmi, F., Glitho, R., Crespi, N., Morrow, M., Polakos, P., 2016. Wireless sensor network virtualization: a survey. IEEE Commun. Surv. Tutor. 18 (1), 553–576.

Kim, H., Reich, J., Gupta, A., Shahbaz, M., Feamster, N., Clark, R., 2015. Kinetic: verifiable dynamic network control. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, Ser. NSDI15. Plus 0.5em Minus 0.4em. USENIX Association, Berkeley, CA, USA, pp. 59–72.

Klti, R., Kotronis, V., Smith, P., Oct 2013. Openflow: a security analysis. In: 2013 21st IEEE International Conference on Network Protocols (ICNP), pp. 1–6.

Kobo, H.I., Abu-Mahfouz, A.M., Hancke, G.P., 2017. A survey on software-defined wireless sensor networks: challenges and design requirements. IEEE Access 5, 1872–1899.

Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., Shenker, S., Onix, 2010. A distributed control platform for large-scale production networks. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, Ser. OSDI10. Plus 0.5em Minus 0.4em. USENIX Association, Berkeley, CA, USA, pp. 351–364.

Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Ingram, P., Jackson, E., Lambeth, A., Lenglet, R., Li, S.-H., Padmanabhan, A., Pettit, J., Pfaff, B., Ramanathan, R., Shenker, S., Shieh, A., Stribling, J., Thakkar, P., Wendlandt, D., Yip, A., Zhang, R., 2014. Network virtualization in multi-tenant datacenters. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). USENIX Association, Seattle, WA, pp. 203–216. plus 0.5em minus 0.4em.

Kreutz, D., Ramos, F.M.V., Verissimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S., Jan 2015. Software-defined networking: a comprehensive survey. Proc. IEEE 103 (1), 14–76.

Lakshman, A., Malik, P., Apr. 2010. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44 (2), 35–40.

Layeghy, S., Pakzad, F., Portmann, M., Dec 2016. Scor: constraint programming-based northbound interface for sdn. In: 2016 26th International Telecommunication Networks and Applications Conference (ITNAC), pp. 83–88.

Lazaris, A., Tahara, D., Huang, X., Li, E., Voellmy, A., Yang, Y.R., Yu, M., Tango, ACM, 2014. Simplifying sdn control with automatic switch property inference, abstraction, and optimization. In: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, Ser. CoNEXT 14. Plus 0.5em Minus 0.4em, pp. 199–212 New York, NY, USA.

Li, Y., Chen, M., 2015. Software-defined network function virtualization: a survey. IEEE Access 3, 2542–2553.

Li, Y., Su, X., Riekki, J., Kanter, T., Rahmani, R., 2016. A sdn-based architecture for horizontal internet of things services. In: Communications (ICC), IEEE International Conference. Plus 0.5em Minus 0.4em. IEEE, pp. 1–7.

Li, X., Wang, X., Liu, F., Xu, H., Dhl, July 2018. Enabling flexible software network functions with fpga acceleration. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 1–11.

Lin, P., Hart, J., Krishnaswamy, U., Murakami, T., Kobayashi, M., Al-Shabibi, A., Wang, K.-C., Bi, J., Aug. 2013. Seamless interworking of sdn and ip, *SIGCOMM Comput*. Commun. Rev. 43 (4), 475–476.

Lin, P., Bi, J., Hu, H., July 2014. Btsdn: bgp-based transition for the existing networks to sdn. In: 2014 Sixth International Conference on Ubiquitous and Future Networks (ICUFN), pp. 419–424.

Lin, P., Bi, J., Chen, Z., Wang, Y., Hu, H., Xu, A., April 2014. We-bridge: west-east bridge for sdn inter-domain network peering. In: Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on, pp. 111–112.

Lu, J., Zhang, Z., Hu, T., Yi, P., Lan, J., 2019. A survey of controller placement problem in software-defined networking. IEEE Access 7, 24290–24307.

Luo, T., Tan, H.P., Quek, T.Q.S., November 2012. Sensor openflow: enabling software-defined wireless sensor networks. IEEE Commun. Lett. 16 (11), 1896–1899.

Majidi, A., Gao, X., Zhu, S., Jahanbakhsh, N., Chen, G., 2019. Adaptive routing reconfigurations to minimize flow cost in sdn-based data center networks. In: Proceedings of the 48th International Conference on Parallel Processing. Plus 0.5em Minus 0.4em. ACM, p. 50.

E. Mannie, Generalized multi-protocol label switching (gmpls) architecture, 2004, Accessed on: 01-Jan-2020. [Online]. Available: https://www.ietf.org/rfc/rfc3945.txt.

Marconett, D., Yoo, S.J.B., Flowbroker, Apr 2015. A software-defined network controller architecture for multi-domain brokering and reputation. J. Netw. Syst. Manag. 23 (2), 328–359.

Masoudi, R., Ghaffari, A., 2016. Software defined networks: a survey. J. Netw. Comput. Appl. 67, 1–25.

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

Migration working group, Accessed on: 01-Jan-2020. [Online]. Available: https://www.opennetworking.org/tag/migration-working-group/.

Mijumbi, R., Serrat, J., Gorricho, J.-L., Bouten, N., Turck, F.D., Boutaba, R., 2016. Network function virtualization: state-of-the-art and research challenges. IEEE Commun. Surv. Tutor. 18 (1), 236–262.

Monsanto, C., Foster, N., Harrison, R., Walker, D., Jan. 2012. A compiler and run-time system for network programming languages. SIGPLAN Not 47 (1), 217–230.

Monsanto, C., Reich, J., Foster, N., Rexford, J., Walker, D., 2013. Composing software-defined networks. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, Ser. Nsdi13. Plus 0.5em Minus 0.4em. USENIX Association, Berkeley, CA, USA, pp. 1–14.

Mottola, L., Picco, G.P., Apr. 2011. Programming wireless sensor networks: fundamental concepts and state of the art. ACM Comput. Surv. 43 (3), 19 119:51.

Nascimento, M.R., Rothenberg, C.E., Salvador, M.R., Corra, C.N.A., de Lucena, S.C., Magalhes, M.F., 2011. Virtual routers as a service: the routeflow approach leveraging software-defined networks. In: Proceedings of the 6th International Conference on Future Internet Technologies. ACM, New York, NY, USA, pp. 34–37. ser. CFI 11. plus 0.5em minus 0.4em.

NBIWG, Accessed on: 01-Jan-2020. [Online]. Available: https://www.opennetworking.org/tag/nbi-working-group/.

Nelson, T., Ferguson, A.D., Scheer, M.J., Krishnamurthi, S., 2014. Tierless programming and reasoning for software-defined networks. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Plus 0.5em Minus 0.4emSeattle. USENIX Association, WA, pp. 519–531.

NEMO, 2015. NeMo: An Applications Interface to Intent Based Networks. Accessed on: 10-Feb-2020. [Online]. Available: http://nemo-project.net/.

NEMO, 2020. Netowrk Modeling for ODL Main. Accessed on: 01-Jan-2020. [Online]. Available: https://wiki.opendaylight.org/view/NEMO.

Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G., 2007. Minizinc: towards a standard cp modelling language. In: Bessire, C. (Ed.), In *Principles and Practice of Constraint Programming CP 2007*. Springer Berlin Heidelberg, Heidelberg, pp. 529–543. plus 0.5em minus 0.4emBerlin.

Ojo, M., Adami, D., Giordano, S., Dec 2016. A sdn-iot architecture with nfv implementation. In: 2016 IEEE Globecom Workshops (GC Wkshps), pp. 1–6.

Oktian, Y.E., July 2015. Secure your northbound sdn api. In: 2015 Seventh International Conference on Ubiquitous and Future Networks, pp. 919–920.

ONF, Accessed on: 01-Jan-2020. [Online]. Available: https://www.opennetworking.org/.

ONF-TS-006, 2012, OpenFlow switch specifications Version 1.3, Accessed on: 01-Jan-2020. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

ONF-TS-012, 2013. OpenFlow switch specifications. Accessed on: 01-Jan-2020. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf. Version 1.4.

ONF-TS-016, 2014. OpenFlow management and configuration protocol. Accessed on: 01-Jan-2020. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.2.pdf.

ONF-TS-025, 2015. OpenFlow switch specifications. Accessed on: 01-Jan-2020. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf. Version 1.5.

OpenDaylight: A Linux foundation collaborative project, Accessed on: 01-Jan-2020. [Online]. Available: https://www.opendaylight.org.

OpenDaylight AAA project, Accessed on: 01-Jan-2020. [Online]. Available: https://wiki.opendaylight.org/view/AAA:Main.

Parniewicz, D., Doriguzzi Corin, R., Ogrodowczyk, L., Rashidi Fard, M., Matias, J., Gerola, M., Fuentes, V., Toseef, U., Zaalouk, A., Belter, B., Jacob, E., Pentikousis, K., 2014. Design and implementation of an OpenFlow hardware abstraction layer. In: Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing, Ser. DCC 14. Plus 0.5em Minus 0.4em. ACM, New York, NY, USA, pp. 71–76.

Pfaff, B., Davie, B., December 2013. The open vSwitch database management protocol, informational, internet engineering task force, RFC 7047. [Online]. Available: http://www.ietf.org/rfc/rfc7047.txt.

B. Pfaff and B. Davie, The open vSwitch database management protocol, Accessed on: 01-Jan-2020. [Online]. Available: https://tools.ietf.org/pdf/rfc7047.pdf.

Pham, M., Hoang, D.B., June 2016. Sdn applications - the intent-based northbound interface realisation for extended applications. In: 2016 IEEE NetSoft Conference and Workshops (NetSoft), pp. 372–377.

Phemius, K., Bouet, M., Leguay, J., May 2014. Disco: distributed multi-domain sdn controllers. In: 2014 IEEE Network Operations and Management Symposium (NOMS), pp. 1–4.

Pranata, A.A., Jun, T.S., Kim, D.S., 2019. Overhead reduction scheme for sdn-based data center networks. Comput. Stand. Interfac. 63, 1–15.

Qadir, J., Hasan, O., Firstquarter 2015. Applying formal methods to networking: theory, techniques, and applications. IEEE Commun. Surv. Tutor. 17 (1), 256–291.

Qin, Z., Denker, G., Giannelli, C., Bellavista, P., Venkatasubramanian, N., May 2014. A software defined networking architecture for the internet-of-things. In: 2014 IEEE Network Operations and Management Symposium (NOMS), pp. 1–9.

Quagga routing suite, Accessed on: 01-Jan-2020. [Online]. Available: http://www.nongnu.org/quagga/.

RabbitMQ, 2020. An open source messaging protocol. Accessed on: 01-Jan-2020. [Online]. Available: https://www.rabbitmq.com/.

Reitblatt, M., Canini, M., Guha, A., Foster, N., 2013. Fattire: declarative fault tolerance for software-defined networks. In: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Ser. HotSDN 13. Plus 0.5em Minus 0.4em. ACM, New York, NY, USA, pp. 109–114.

T. L. Y. Rekhter and S. Hares, A border gateway protocol 4 (BGP-4), Accessed on: 01-Jan-2020. [Online]. Available: https://tools.ietf.org/html/rfc4271.

Y. Rodrigues, OpenDaylight ODL: network intent composition (NIC) - a real intent-based solution, challenges and next stepsIntent framework, https://www.serro.com/opendaylight-network-intent-composition-a-real-intent-based-solution-challenges-and-next-steps/, Accessed on: 01-Jan-2020..

Salman, O., Elhajj, I., Kayssi, A., Chehab, A., Nov 2015. An architecture for the internet of things with decentralized data and centralized control. In: 2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA), pp. 1–8.

Samociuk, D., 2015. Secure Communication Between OpenFlow Switches and Controllers. In: The Seventh International Conference on Advances in Future Internet. IARIA.

Santos, M.A.S., Nunes, B.A.A., Obraczka, K., Turletti, T., de Oliveira, B.T., Margi, C.B., Sept 2014. Decentralizing sdn's control plane. In: 39th Annual IEEE Conference on Local Computer Networks, pp. 402–405.

Secci, S., Attou, K., Phung, D.C., Scott-Hayward, S., Smyth, D., Vemuri, S., Wang, Y., 2017. ONOS security and performance analysis. (Report No. 1). Accessed on: 01-Jan-2020. [Online]. Available: https://onosproject.org/wp-content/uploads/2018/01/ONOS-security-and-performance-analysis-brigade-report-no1.pdf.

Secci, S., Scott-Hayward, S., Wang, Y., Van, Q.P., 2018. ONOS security and performance analysis. (Report No. 2). Accessed on: 01-Jan-2020. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2018/11/secperf_report_2.pdf.

Sherwood, R., Chan, M., Covington, A., Gibb, G., Flajslik, M., Handigol, N., Huang, T.-Y., Kazemian, P., Kobayashi, M., Naous, J., Seetharaman, S., Underhill, D., Yabe, T., Yap, K.-K., Yiakoumis, Y., Zeng, H., Appenzeller, G., Johari, R., McKeown, N., Parulkar, G., Jan. 2010. Carving research slices out of your production networks with OpenFlow. SIGCOMM Comput. Commun. Rev. 40 (1), 129–130.

Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., Parulkar, G., 2010. Can the production network be the testbed? In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, Ser. OSDI10. Plus 0.5em Minus 0.4em. USENIX Association, Berkeley, CA, USA, pp. 365–378.

Siow, E., Tiropanis, T., Hall, W., Jul. 2018. Analytics for the internet of things: a survey. ACM Comput. Surv. 51 (4) 74:174:36.

Smith, M., Dvorkin, M., Laribi, V., Pandey, V., Gerg, P., Weidenbacher, N., OpFlex control protocol. Accessed on: 01-Jan-2020. [Online]. Available: https://tools.ietf.org/html/draft-smith-opflex-03.

Song, H., 2013. Protocol-oblivious forwarding: unleash the power of sdn through a future-proof forwarding plane. In: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Ser. HotSDN 13. Plus 0.5em Minus 0.4em. ACM, New York, NY, USA, pp. 127–132.

Soul, R., Basu, S., Kleinberg, R., Sirer, E.G., Foster, N., 2013. Managing the network with merlin. In: Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, Ser. HotNets-XII. Plus 0.5em Minus 0.4em. ACM, New York, NY, USA. 24:124:7.

Stewart, R., 2007. Stream control transmission protocol. Accessed on: 01-Jan-2020. [Online]. Available: https://tools.ietf.org/html/rfc4960.

Stribling, J., Sovran, Y., Zhang, I., Pretzer, X., Li, J., Kaashoek, M.F., Morris, R., 2009. Flexible, wide-area storage for distributed systems with wheelfs. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, Ser. NSDI09. Plus 0.5em Minus 0.4em. USENIX Association, Berkeley, CA, USA, pp. 43–58.

Su, M., Alvarez, V., Jungel, T., Toseef, U., Pentikousis, K., Sept 2014. An OpenFlow implementation for network processors. In: 2014 Third European Workshop on Software Defined Networks, pp. 123–124.

Tam, A.S.W., Xi, K., Chao, H.J., April 2011. Use of devolved controllers in data center networks. In: 2011 IEEE Conference on Computer Communications Workshops. INFOCOM WKSHPS, pp. 596–601.

Tennenhouse, D.L., Smith, J.M., Sincoskie, W.D., Wetherall, D.J., Minden, G.J., Jan 1997. A survey of active network research. IEEE Commun. Mag. 35 (1), 80–86.

Tootoonchian, A., Ganjali, Y., 2010. Hyperflow: a distributed control plane for OpenFlow. In: Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking, Ser. INM/WREN10. Plus 0.5em Minus 0.4em. USENIX Association, Berkeley, CA, USA. 33.

Tootoonchian, A., Gorbunov, S., Ganjali, Y., Casado, M., Sherwood, R., 2012. On controller performance in software-defined networks. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, Ser. Hot-ICE12. Plus 0.5em Minus 0.4emBerkeley. USENIX Association, CA, USA. 1010.

Trois, C., Fabro, M.D.D., de Bona, L.C.E., Martinello, M., 2016. A survey on sdn programming languages: toward a taxonomy. IEEE Commun. Surv. Tutor. 18 (4), 2687–2712 Fourthquarter.

Tseng, Y., Zhang, Z., Nat-Abdesselam, F., Controllersepa, Dec 2016. A security-enhancing sdn controller plug-in for OpenFlow applications. In: 2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 268–273.

Turull, D., Hidell, M., Sjdin, P., June 2012. libnetvirt: the network virtualization library. In: 2012 IEEE International Conference on Communications (ICC), pp. 5543–5547.

Vasseur, J., Roux, J.L., 2009. Path computation element (pce) communication protocol (pcep). Accessed on: 01-Jan-2020. [Online]. Available: https://www.ietf.org/rfc/rfc5440.txt.

Ventre, P.L., Tajiki, M.M., Salsano, S., Filsfils, C., 2018. SDN architecture and southbound apis for ipv6 segment routing enabled wide area networks. CoRR abs/1810.06008. [Online]. Available: http://arxiv.org/abs/1810.06008.

Voellmy, A., Hudak, P., 2011. Nettle: Taking the sting out of programming network routers. In: Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages, Ser. PADL11. Plus 0.5em Minus 0.4em. Springer-Verlag, Berlin, Heidelberg, pp. 235–249.

Voellmy, A., Kim, H., Feamster, N., 2012. Procera: a language for high-level reactive network control. In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, Ser. HotSDN 12. Plus 0.5em Minus 0.4em. ACM, New York, NY, USA, pp. 43–48.

Wang, L., Lu, Z., Wen, X., Knopp, R., Gupta, R., 2016. Joint optimization of service function chaining and resource allocation in network function virtualization. IEEE Access 4, 8084–8094.

Wang, J., Shou, G., Hu, Y., Guo, Z., Oct 2016. A multi-domain sdn scalability architecture implementation based on the coordinate controller. In: 2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), pp. 494–499.

Wang, T., Xu, H., Liu, F., June 2017. Multi-resource load balancing for virtual network functions. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 1322–1332.

Wang, T., Liu, F., Xu, H., Oct 2017. An efficient online algorithm for dynamic sdn controller assignment in data center networks. IEEE/ACM Trans. Netw. 25 (5), 2788–2801.

Weng, J., Weng, J., Zhang, Y., Luo, W., Lan, W., Jan 2019. Benbi: scalable and dynamic access control on the northbound interface of sdn-based vanet. IEEE Trans. Veh. Technol. 68 (1), 822–831.

Wibowo, F.X.A., Gregory, M.A., Dec 2016. Software defined networking properties in multi-domain networks. In: 2016 26th International Telecommunication Networks and Applications Conference (ITNAC), pp. 95–100.

Wibowo, F.X., Gregory, M.A., Ahmed, K., Gomez, K.M., 2017. Multi-domain software defined networking: research status and challenges. J. Netw. Comput. Appl. 87 (Supplement C), 32–45.

Xu, Z., Liu, F., Xu, H., June 2016. Demystifying the energy efficiency of network function virtualization. In: 2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS), pp. 1–10.

Yamanaka, H., Kawai, E., Ishii, S., Shimojo, S., Sept 2014. Autovflow: autonomous virtualization for wide-area OpenFlow networks. In: 2014 Third European Workshop on Software Defined Networks, pp. 67–72.

YangUI, 2020. OpenDaylight DLUX: YangUI-user. Accessed on: 01-Jan-2020. [Online]. Available: https://wiki.opendaylight.org/view/OpenDaylight_dlux:yangUI-user.

Yang, H., Zhang, J., Zhao, Y., Ji, Y., Wu, J., Han, J., Lee, Y., 05 2015. Performance evaluation of multi-stratum resources integrated resilience for software defined inter-data center interconnect. Optic Express 23, 13384.

Yang, H., Zhang, J., Zhao, Y., Han, J., Lin, Y., Lee, Y., Sudoi, February 2016. Software defined networking for ubiquitous data center optical interconnection. IEEE Commun. Mag. 54 (2), 86–95.

Yangyang, W., Jun, B., Survey of mechanisms for inter-domain SDN. Accessed on: 01-Jan-2020. [Online]. Available: https://www.zte.com.cn/global/about/magazine/zte-communications/2017/3/en_225/465746.

Yap, K.-K., Huang, T.-Y., Dodson, B., Lam, M.S., McKeown, N., 2010. Towards software-friendly networks. In: Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems, Ser. APSys 10. Plus 0.5em Minus 0.4em. ACM, New York, NY, USA, pp. 49–54.

Yazici, V., Sunay, M.O., Ercan, A.O., 2014. Controlling a software-defined network via distributed controllers. CoRR abs/1401.7651.

Yin, H., Xie, H., Tsou, T., Lopez, D., Aranda, P., Sidi, R., SDNi, June 2012. A Message Exchange Protocol for Software Defined Networks (SDNS) across Multiple Domains. Internet Draft, Internet Engineering Task Force.

Yu, M., Wundsam, A., Raju, M., Nosix, Apr. 2014. A lightweight portability layer for the sdn os. SIGCOMM Comput. Commun. Rev. 44 (2), 28–35.

Yu, H., Li, K., Qi, H., Li, W., Tao, X., Sept 2015. Zebra: an east-west control framework for sdn controllers. In: 2015 44th International Conference on Parallel Processing, pp. 610–618.

ZebOS, Accessed on: 01-Jan-2020. [Online]. Available: http://www.ipinfusion.com/products/zebos/.

Zeng, C., Liu, F., Chen, S., Jiang, W., Li, M., April 2018. Demystifying the performance interference of co-located virtual network functions. In: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, pp. 765–773.

Zhu, L., Karim, M.M., Sharif, K., Li, F., Du, X., Guizani, M., 2019. SDN Controllers: Benchmarking & Performance Evaluation, vol. abs/1902. *CoRR* 04491. [Online]. Available: http://arxiv.org/abs/1902.04491.

Zohaib Latif did his BS in Electrical Engineering in 2006 and MS in Electrical and Electronics Engineering from University of Glasgow, UK in 2008. Since 2011 he was working as senior lecturer and is currently a final year PhD scholar at School of Computer Science, Beijing Institute of Technology, Beijing, China. His major interests are in Software Defined Networks (SDN), Distributed Controllers in SDN, and Internet of Things.

Kashif Sharif (M′08) received his MS degree in information technology in 2004, and PhD degree in computing and informatics from University of North Carolina at Charlotte, USA in 2012. He is currently an associate professor for research at Beijing Institute of Technology, China. His research interests include information centric networks, blockchain & distributed ledger technologies, wireless & sensor networks, software defined networks, and data center networking. He also serves as associate editor for IEEE Access.

Fan Li (M′12) received the PhD degree in computer science from the University of North Carolina at Charlotte in 2008, MEng degree in electrical engineering from the University of Delaware in 2004, MEng and BEng degrees in communications and information system from Huazhong University of Science and Technology, China in 2001 and 1998, respectively. She is currently a professor at School of Computer Science in Beijing Institute of Technology, China. Her current research focuses on wireless networks, ad hoc and sensor networks, and mobile computing. Her papers won Best Paper Awards from IEEE MASS (2013), IEEE IPCCC (2013), ACM MobiHoc (2014), and Tsinghua Science and Technology (2015). She is a member of ACM and IEEE.

M. M. Karim is pursuing his Ph.D. in Computer Science and Technology at Beijing Institute of Technology, Beijing, China. Previously, he received both M.Eng and B. Eng in Computer Science from Northwestern Polytechnical University, Xi'an, China. His research interests include Software-Defined Networking, Information-Centric Networking, Named Data Networks, and Next-Generation Networking

Sujit Biswas (GS′17) is enrolled as PhD fellow in Beijing Institute of Technology, China. He received his M.Sc. degree in Computer Engineering from Northwestern Polytechnical University, China in 2015. He is also an Assistant Professor with Computer Science and Engineering department, Faridpur Engineering College, University of Dhaka, Bangladesh. His basic research interest is in IoT, Blockchain, Mobile computing security and privacy.

Yu Wang (F′18) is a Professor of Computer Science at the University of North Carolina at Charlotte. He holds a Ph.D. from Illinois Institute of Technology, an MEng and a BEng from Tsinghua University, all in Computer Science. His research interest includes wireless networks, smart sensing, and mobile computing. He has published over 200 papers in journals and conferences, with four best paper awards. He has served as Editorial Board Member of several international journals, including IEEE Transactions on Parallel and Distributed Systems. He is a fellow of IEEE, a senior member of ACM, and a member of AAAS.