Wireless Networks
https://doi.org/10.1007/s11276-022-03044-3

ORIGINAL PAPER

Performance evaluation of Software-Defined Network (SDN) controllers using Dijkstra's algorithm

Yinjun Zhang[1] · **Mengji Chen**[2]

Accepted: 1 June 2022

Abstract
The advancement of technology, specifically the development of information technology (IT) has had a great influence on everyone life. Due to the rapid development of technology, the traditional network architectures are no longer adequate for the nowadays communication, education, businesses, carriers, and end-users, etc. This indeed is a serious issue, as every organization, enterprise, and educational institute, etc. wants to keep itself on top and to facilitate their customers as much as possible by providing high quality services with minimum delay, etc. In order to tackle this problem, a new emerging network architecture known as software defined networking (SDN) may be employed, as it is more interactive, more flexible, controllable, scalable, and programmable. A network system has two main planes known as control and data planes. The SDN architecture/design separates the control and data planes, legalizes network information and state, and keeps the network infrastructure out of the applications. In contrast, a network may reach a point where the computer or network resources restrict the data flow governed by the bandwidth. In SDN architecture, the controller is the most significant and central component, and large SDN networks might have numerous controllers or controller domains that share network administration. Due to the obvious importance of controllers, different studies have been conducted to compare, test, and assess their performance. This study examines the implementation of Dijkstra's algorithm using two of the most important SDN open-source controllers (POX and RYU), which permits packet acquisition and transmission between end devices via the network's shortest and/or lowest load pathways. Performance of the two utilized controllers is measured in terms of various quality of service (QoS) metrics such as throughput, packet delivery ratio, jitter, and packet loss on a specific network architecture under a variety of workloads. Further, we developed a customized network topology using the Mininet which is an emulator tool for SDN. In addition, we also measured the performance of the utilized controllers in terms of various QoS metrics, using the Mininet tool. The experimental results show that the proposed system attained promising results in terms of all QoS metrics.

Keywords SDN; SDN controllers; Dijkstra's algorithm · Shortest Route; Open Flow · Mininet · POX · RYU · Jitter · Throughput · Packet delivery ratio

1 Introduction

With the increasing number of mobile devices and usage of mobile devices, virtualization, and increasing prevalence of cloud computing have caused network architects to rethink about the traditional structures and layouts of the networks. In recent times, the data centers have undergone significant transformations.

The current network architecture of different enterprises uses multiple databases and servers, which results in continuous machine-to-machine (M2M) communication, causing heavy traffic between the communicating devices. Further, the network traffic sequences changes when the consumers access the content from different network devices. In the last decade, data centers have grown significantly in order to satisfy the ever-changing needs. The design and architecture

✉ Mengji Chen
mengji001@hcnu.edu.cn

Yinjun Zhang
zhangyinjun@gxstnu.edu.cn

[1] School of mechanical and electrical engineering, Guangxi Science and Technology Normal University, Guangxi, China

[2] School of Artificial Intelligence and Smart Manufacturing, Hechi University, Yizhou, Guangxi, China

Published online: 20 August 2022

of the data centers get increasingly complicated, as the traffic grows across large area networks. With the rising usage of mobile networks in accessing devices such as smartphones, tablets, laptop computers, etc., it becomes more important to manage not only traffic but also continuity in all contact sectors (i.e., speed, security). On the basis of the foregoing, it is quite obvious to conclude that, the traditional network architectures are incapable of meeting the modern network requirements. The present infrastructure cannot fulfill the traffic, scalability, and availability demands of consumers, organizations, and vendors due to a number of reasons. Several protocols are used in network communication to connect nodes over long distances with varying speed, topology, and service specifications. Routers, multiple switches, firewalls, network authentication gateways, and other hardware devices used for communication purposes must be configured when installing or uninstalling a system [1]. The current network architectures are known to be relatively static for all of the factors mentioned above. On the other side, server virtualization has greatly increased the number of nodes needing network access. Applications are also installed on virtual computers that enables data transmission and dataflow among the connected devices. Further, many businesses uses an IP-converging network for voice, data, and video traffic. Despite the fact that, the current networks have different service quality levels for different applications, such services are manual. The network has not been able to keep up with changes in app traffic and customer needs.

1.1 Traditional network architectures

Nowadays, networks are physically separated to meet the needs of organizations, industries, end-users, and service providers, with the control plane and data forwarding on the same device. Even if this network design approach has functioned adequately in the past, the conventional network would find it difficult, if not inconceivable, to satisfy the recent virtualized needs. Corporate information management teams are insisting on virtualizing the majority of their servers due to the availability of limited budget. Such a process is difficult because the demand for both user mobility and application is increasing [2].

1.2 Conventional networks constraints

Despite the fact that, the current conventional network architectures were not designed to meet the nowadays enterprises, end-users, and service providers' requirements. The traditional network design has various limitations and are given as follows:

1.2.1 Management complexity

Previously, network architectures were based on a combination of routing protocols that were intended to successfully link hosts across large distances at high speeds and in a variety of network configurations. During the last few decades, protocols have been developed in a variety of ways to meet market criteria such as high protection, availability, and increased communication, resulting in separation, where each protocol addresses a specific form of the issue without considering abstractions. Nowadays, one of the major issues that the network administrators are facing is the network management complexity. For example, adding or removing a network device, becomes inconvenient for the network administrators because several aspects of the network needs reconfiguration, including routing protocols, VLANs, access lists, network topologies, and QoS policies. Aside from the aforementioned, before making any network changes, infrastructure manufacturer and device release reliability must be addressed. As a consequence, managers retain their networks basically unaltered to avoid or reduce service interruptions associated with any transformation.

The static nature of network architecture limits the dynamic nature of server virtualization, which in turn expands the number of hosts that require connectivity. Virtualization services allows a single server to connect to a subset of clients. Although, thanks to virtualization's resources, it is now possible to distribute programs across multiple virtual machines that are linked together. In many cases, VMs must be transited in order to achieve balanced workloads; this feature of virtualized networks puts a lot of strain on traditional networking designs, which were not designed to handle such a complex transitions flow [3].

1.2.2 Difficulty in applying policies

The network administrators must configure all of the networking devices (many switches, gateways, and routers) and communication tools in order to maintain an enterprise network strategy. The advancement of IT has brought a revolution in the field of networking. As it connects a virtual machine (VM) to the network in virtualization, and the network administrator needs to install and change access control lists (ACLs) around the entire network. With today's network sophistication, network administrators are finding it increasingly difficult to maintain clear settings for access rights, protection and quality of services [4, 5].

1.2.3 Scalability issues

Data centers typically have a strong demand to expand exponentially due to the rapid change in the number of

connected devices, resulting in the expansion of the network at the same time. Historically, network administrators have depended primarily on bandwidth excess demand to grow business networks, but traffic flows have become increasingly complex as a result of data center virtualization, making traffic prediction impossible. Manually configuring such large-scale networks is indeed a very tough task [6, 7].

1.2.4 Equipment manufacture dependability

In response to the current market trends and increasing demand of the customers, the Internet service providers (ISPs) and data centers continue to look for ways to expand their capabilities and offerings in order to fulfill the demands of the customers according to the revolution of technology. The equipment vendor's life cycle for the supplied service and equipment, which can be two years or more in some cases, limits the ability to react [8]. Further, the lack of open and generic interfaces has hampered the network operators' ability to adapt the network to their unique environments. The industry has suffered as a result of the disconnection between the network capability and requirements of the customers.

2 Software defined networking (SDN) overview

SDN principles may be traced back to the separation of the control and data planes, which was initially utilized in the public switched telephone network to facilitate provisioning and management long before it was adopted by data networks. SDN has the potential to transform how network engineers and designers manage and develop their networks in order to meet business requirements. Networks becomes open standard, non-proprietary, simple to program, and easy to maintain with the introduction of SDN. SDN is one of the most popular models used to overcome the limitations of the conventional computer network designs in order to satisfy the current and dynamic network requirements. The SDN concept separates the control plane (the network's brain) from the data plane.

The following subsections discusses in more detail about the background of SDN, benefits of SDN, architecture of SDN, and related literature of the SDN.

2.1 Background of SDN

When the OpenFlow protocol and SDN technology were launched in 2011, there was a lot of buzz. Since that time, adoption has been gradual, particularly among smaller businesses with less resources and network infrastructure. The concept of SDN was inspired by the need to enable user-controlled forwarding management in network nodes. It is important to note that the idea of programmable networks and the segmentation of control and data planes has been around for a while. The earlier programmable networking projects are summarized in this section, and are presented with some background in the following subsections.

2.1.1 The Open Signaling (OPENSIG)

This group of experts was started in 1995 with a number of seminars aiming to "improve the ATM, Internet, mobile network accessibility, expandability, and programmability" [9]. At that time the hardware and software used in the communication process were required to be separated but due to the availability of limited resources it was difficult to do so at that time. Vertically integrated routers and switches are primarily responsible for this. In addition to the closed nature of these switches and routers, the rapid introduction of a new core proposal included the provision of network hardware connections via free, programmable network interfaces, allowing the use of new services via a scattered programming environment.

2.1.2 General switch Management Protocol (GSMP)

The GSMP protocol defined how switches could be handled, and it was created by the Internet Engineering Task Force (IETF). Among other things, GSMP assists the controller in establishing and terminating the transfer links, adding and removing multicast relationships, managing the switch ports, requesting configuration information, and requesting and terminating switch resource reservations [10, 11]. In June 2002, the working group was formally disbanded, and the current standard suggestion (GSMPv3) was published.

2.1.3 Active networking

Another program known as active networking, which began in the mid-1990s, proposed a network architecture concept that could be configured to provide customized services [12]. In this program the two key techniques were considered such as, (a) programmable user switch with in-band data transmission and out-of-band control channels; (b) capsules, which were programming segments that could be sent in user messages and then translated and run by the routers.

2.1.4 Tempest

Tempest is a platform for secure and programmable networks that was proposed in 1998. The Tempest architecture provides a programmable network environment by allowing

the implementation and updating of two-level networks. The Tempest structure also enables facilities to be refined at an expert level of granularity via the relation termination principle. The Tempest system's characteristics also permits the service providers in order to become network operators for individual physical network partitions. This enabled them to put their knowledge of how network components work by programming their own control scheme [13].

2.1.5 Path computation element (PCE)

The Path Computation Element Protocol (IETF standard), which was established in 2004, is a control protocol that runs on MPLS networks and eliminates the need for routers to describe network traffic. The PCE framework, as defined in RFC 4655 (2006), optimizes route calculation by decoupling network topology from route creation.

2.1.6 The 4D project

The 4D Project, which began in 2004, demonstrate the separation of routing decision logic protocols. The "decision" plane would have had a global network perspective, and facilities from the "dissemination" and "discovery" planes would have been required to monitor a data plane for traffic transmission. The ability of the network controller to learn about the available resources is referred to as "discovery". The process of determining the network's topology is referred to as dissemination. Similar concepts influenced later initiatives, such as NOX, which proposed an operating system for networks using OpenFlow connectivity [14].

2.1.7 Network Configuration Protocol (NetConf)

The IETF Network Configuration Working Group proposed the NETCONF as a control protocol for changing the configuration of network equipment in 2006 [15]. NETCONF made the API accessible to network devices, allowing for the transmission and extraction of all configuration data. Because a NETCONF network should not be considered completely programmable, it is critical to incorporate all new features at both the network interface and the administrator level to allow for careful maintenance. NETCONF is primarily intended to facilitate automated setup without granting direct access to the government data.

2.1.8 Forwarding and control element separation (ForCES)

This working group chose a path that is similar to the working of SDN. ForCES is an internal network interface design that separates the network control and forwarding functions. Outside of the universe, however, the combined entity is always referred to as a single network unit [16]. The transmitting and controlling elements of the ForCES network communicate with one another via the ForCES protocol. The integrated entity (forwarding and control elements) is now regarded as a single external network feature, in contrast to the SDN architecture. Further, this project was completed in 2015.

2.1.9 Ethane

In 2006, the SANE/Ethane framework, which provided a new framework for business networks, was a forerunner to OpenFlow. Ethane emphasized the use of a centralized administrator to handle network security and regulation. The provision of identity-based access management is a good example of this framework. Ethane had two main components, similar to SDN: a controller that determined when a packet could be transmitted and a network [17]. A routing table and a secured connection to the controller was included in the transmission of ethane. Ethane was also instrumental in laying the groundwork for SDN. Ethane's identity-based access management may be used as a service on top of an SDN controller like NOX, SNAC, Maestro, Beacon, or Helios, among others, to fit into the modern SDN paradigm.

2.1.10 Open networking Foundation (ONF)

The ONF is a user-driven group devoted to establishing and deploying SDN via an open standards in areas where such standards are vital to the networking industry's progress. The ONF is a non-profit dedicated platform to the development of open protocols such as the OpenFlow protocol standard for configuration and management [18]. The OpenFlow protocol is the first standard supplier communication protocol specified between the controls and forwarding layers of an SDN paradigm.

2.2 Benefits of SDN

SDN would increase network flexibility for businesses and enterprises, allowing them to customize and refine their networks while lowering total network management costs. Some of the key benefits of SDN are given as follows;

- ***Simplicity in Network Management***: The SDN can display and control the network as a single node, transfer to an easy interface to handle complex network maintenance functions.
- ***Deployment of Quick Service***: New features and applications can now be launched in a matter of hours rather than days.

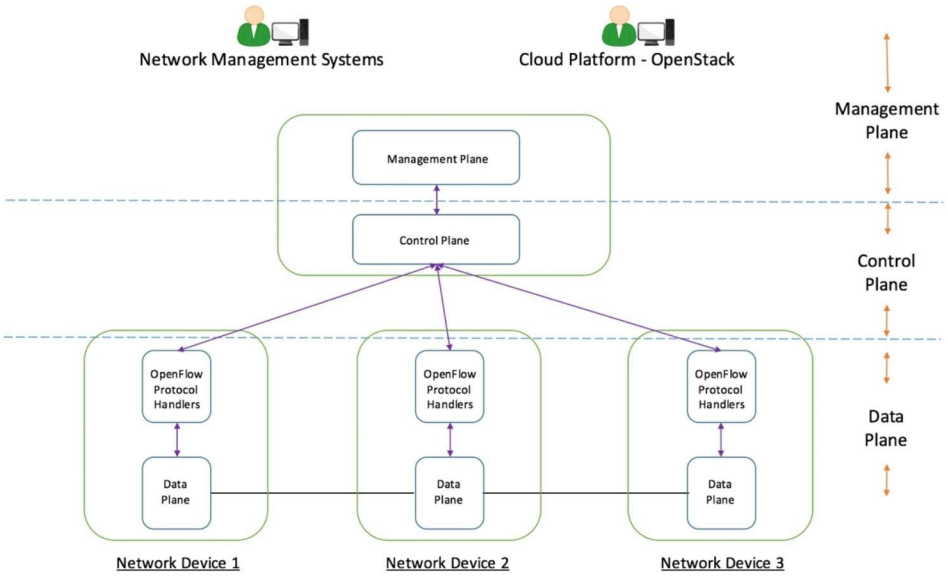

Fig. 1 SDN Architecture

- ***Automated Configuration***: VLAN assignment and QoS configuration are examples of manually configured functions that can be dynamically provisioned.
- ***Network Virtualization***: Although virtualization of servers and storage is more prevalent than before, SDN can also virtualize the networks.
- ***Reduction of Running Costs***: It is necessary to improve the network implementation and to make the network automated. The implementation of network architecture has never been simpler leading to more operating cost, however, the implementation of SDN has resulted in lower network operating costs.

2.3 SDN Architecture

The most important component of SDN is the segregation of the forwarding and control planes. The OpenFlow protocol is used by the controller to communicate with the physical or virtual switches. The three main planes (data, control, and application/management) are correctly regarded for the objective of this study. Figure 1 shows a short description the SDN architecture. The three main planes of SDN are discussed with some explanation in the following subsections.

2.3.1 Data plane

The Data Plane (also called as the Forwarding Plane or Data Route) is in-charge of ensuring that the data path packets are managed in compliance with the instructions of the control plane. It is a network architectural plane that physically manages traffic depending on the Control Plane's parameters. It is important for that network segment which forwards user traffic. The rules of the control plane govern forwarding. The other functions of the data plane are buffering, filtering, packet measurement, and so on. Control plane services and applications are frequently terminated in the data plane. Thus, it results in multicasting which is a subset of forwarding, in which the incoming packet must be repeated before being sent to the various output ports [19].

2.3.2 Control plane

The control plane is in-charge of deciding how one or many network nodes should forward packets and communicating these decisions to the network nodes. The traffic routing and network topology are defined by the control plane, which is a network architectural component. It is typically concerned with the forwarding plane as well as the device's operations and maintenance plane. The information on the operational plane may be of interest to the control plane. Management functions are frequently employed in the control plane (in some cases, these make up a separate plane). This relates to network system management, deployment, and repair, as well as decision-making on the status of a network system. The forwarding plane may be downloaded and installed using the administration plane [20].

2.3.3 Application/Management plane

The application plane is where program and utilities that govern network behavior are located. It is responsible for overall network setup, monitoring, and management throughout all network layers architecture. Applications

Table 1 Related literature on SDN

Controller	Methodology	Output
ONOS Open Daylight POX Libfluid Beacon [21]	Network performance characteristics like throughput and end-to-end latency were investigated for the five well-known SDN controllers.	The Libfluid controller had the highest delay compared with other SDN controllers, and the lowest for the ONOS controller.
Beacon Floodlight Open-MUL Open IRIS [22].	Comparative study of the SDN controllers on the basis of latency and bandwidth network QoS performance parameters.	Floodlight is the best controller on basis on basis of latency and Beacon is best in terms of bandwidth.
NOX Beacon Maestro Open Contrail NOX-MT [23]	Finding the optimal and efficient SDN open-source controller based on network topology.	NOX and Beacon controller performance is better than other controllers.
POX Floodlight [25].	The performance of SDN controllers Foodlight and POX was analyzed using Mininet as a tool. The features of the popular SDN controllers are analyzed and its functions are determined.	Floodlight much faster than POX and occupies more memory space for execution to its inbuilt java file.
Floodlight Open Daylight [26].	A load balancing algorithm for SDN underwent a comparative analysis. The use of a centralized controller has proven to be an excellent way to improve and optimize the services.	Floodlight is better than Open Daylight in the term of maximize bandwidth and minimize response times.
POX [28].	Comparison of shortest job algorithm like Round Robin & Random Algorithm implementation in SDN POX controller to find out well known shortest route algorithm for SDN controller.	Due to Round Robin distribute load reasonable based on response time and transaction per second.
Open Daylight [29].	The weighted round-robin (WRR) method is used to implement a distributed load balancing algorithm. The efficiency of the proposed method in the SDN controller is evaluated using throughput and delay network metrics.	In comparison to a traditional SDN technique, the results reveal that a distributed algorithm technique is far more successful (i.e., without load balancing).

plane that explicitly aid in the operation of the forwarding plane are not known to be part of the application plane. It communicates with the controller through a northbound API interface. It is necessary to use these application providers, to customize the flows that will be redirected based on network changes.

2.4 Related literature on SDN

This part provides an overview of comparative studies on the performance of SDN controllers. To examine latency and throughput measures, the majority of systems employ benchmarking or other Cbench tools. It is important to remember that in some literary situations, performance and scalability are similar. As a result, several of the following insights also apply for scalability, which is described in more detail in Table 1.

We discussed the previous research work related to the SDN Controllers, which uses a shortest route selection approach, but the most basic route problems are still there in the POX and RYU controllers. The main objective of this study is to investigate the impact of Dijkstra's algorithm in finding the shortest route using two different SDN controllers (POX and RYU). Further, this study aims to find the best controller for applying Dijkstra's algorithm based on traffic behavior and network performance. It will provide practical simulations of actual network behaviors and provide end-to-end network performance between various SDN controllers and Dijkstra's algorithm after measuring the network performance of Dijkstra's algorithm using SDN open-source controllers.

3 Proposed methodology

Since the use of SDN in different network topologies has begun in recent times, so, a basic understanding of this topic is of a great interest and significance. SDN controllers are built with smaller, highly correlated sets of servers, making them compatible with distributed algorithms that preserve standardized versions of network-wide architectures such as traffic statistics, topology, and so on. SDN does not specify how controllers should be implemented. It could be used to run a variety of network algorithms, from simple ones like shortest path routing to much more complex ones like traffic engineering [30]. SDN controllers manage the entire network, but they must occasionally change the rules for different switches. Despite the fact that the network includes a mix of different and current configuration regulations during the transformation, these principles have the value of handling any provided packet in accordance with a single version. Similar mechanisms can be used to assess per-flow performance.

3.1 OpenFlow a SDN Protocol

OpenFlow (OF) is a protocol that is primarily used and implemented in the SDN. OF is a networking protocol designed specifically for the SDN, allowing the SDN controller to communicate directly with the data forwarding plane and network equipment such as routers and switches in both virtual and physical networks [32]. Further, the SDN controllers manage switches via the OpenFlow protected

channel protocol. If a computer in the SDN needs to connect to a network component, it can do so via the OpenFlow network. OpenFlow is used in the SDN to monitor the hardware by providing routing instructions [33]. Figure 2 illustrates the proposed method.

3.2 Dijkstra's Algorithm

The routing algorithm is responsible for the majority of the problems that exist in existing networks that prevent proper load balancing. The present routing system employs the shortest route algorithm, which requires each packet to seek out the shortest path possible, which is the same for all packets, even if alternate routes are longer but quicker. This actually decreases the network's useable characteristics, such as congestion on the shortest route link, abuse of the same connections, and other network capabilities that aren't regularly utilized [36]. The shortest single-source route is what the Dijkstra algorithm is known for. It determines the shortest path from the source to each of the remaining vertices in the graph as shown in Algorithm 1.

Algorithm 1: DIG-RYU-POX	
Problem: shortest route finding	
Input: Number of (k), all possible path, S, D	
Output: The best path from S to D using controller	
1	Start
2	Connectivity matrix G (I, J)
3	Network matrix C (S, D)
4	INF = all possible path
5	If (Failure > 0)
6	Remove spam node between S and D
7	G (I, j) = possible, matrix = 0
8	End if
9	Dijkstra (K, S, D, G, C)
10	While (N < k)
11	Dijkstra (S, D, G, C)
12	Save all possible path
13	Subtract connectivity matrix
14	N++
15	End while
16	For n pass to controllers
17	For possible path in all paths
18	For S, D, all paths
19	Adjacency [S], [D]
20	End For
21	End For
22	End For

3.3 SDN Controllers

The SDN Controller is a functional object that receives orders or specifications from the SDN application layer and relays them to network devices. In addition, the controller will collect network data from hardware devices and send it to SDN applications. The SDN controller is the control plane for the SDN and has full control over the data plane. The internal design and execution of an SDN controller are not entirely specified by the SDN architecture. It could be a single process or a collection of related processes grouped together to share workloads or avoid failure. It could also be a collection of shared functional elements. Controller elements, like computing services attached to a physical network element, can be run on computer platforms. They can often run on distributed infrastructure, such as virtual machines, in data centers. It's critical to note that the SDN controller is understood to have a global scope, and that its components are understood to communicate with the controller about their data and location. RYU, NOX, POX, Floodlight, and Open Daylight are just a few of the open-source SDN controllers available for designing SDN platforms, but we will focus on two of them: RYU and POX illustrates below in Table 2.

3.4 RYU Controller

The Japanese word RYU means "flow." "Ree-yooh" is how it's pronounced. It's also a Python-based OpenFlow controller with a powerful API that allows developers to program, track, and manage applications. RYU also allows developers to create new network management and control applications by exposing software modules with well- defined APIs. The fact that RYU supports multiple southbound APIs protocols for controlling devices, such as Network Configuration Protocol (NETCONF), OpenFlow Configuration and Management Protocol (OD-Config), and others, is one of its many strengths. The OpenFlow open- source protocol, whose code is available under an Apache License [33], can be used to design network devices based on RYU applications.

3.5 POX Controller

POX is an open-source OpenFlow controller written in Python that is frequently utilized in the academic and research communities due to its ease of usage. It offers a framework for quick prototyping and development of network systems and control apps on the OpenFlow network. POX may be linked to Mininet remotely, and different applications like firewalls, load balancers, vulnerability scanning, routing, and switching can be set up on the core Open Flow switches. POX is integrated into the Mininet and is available for download from its GitHub source [35].

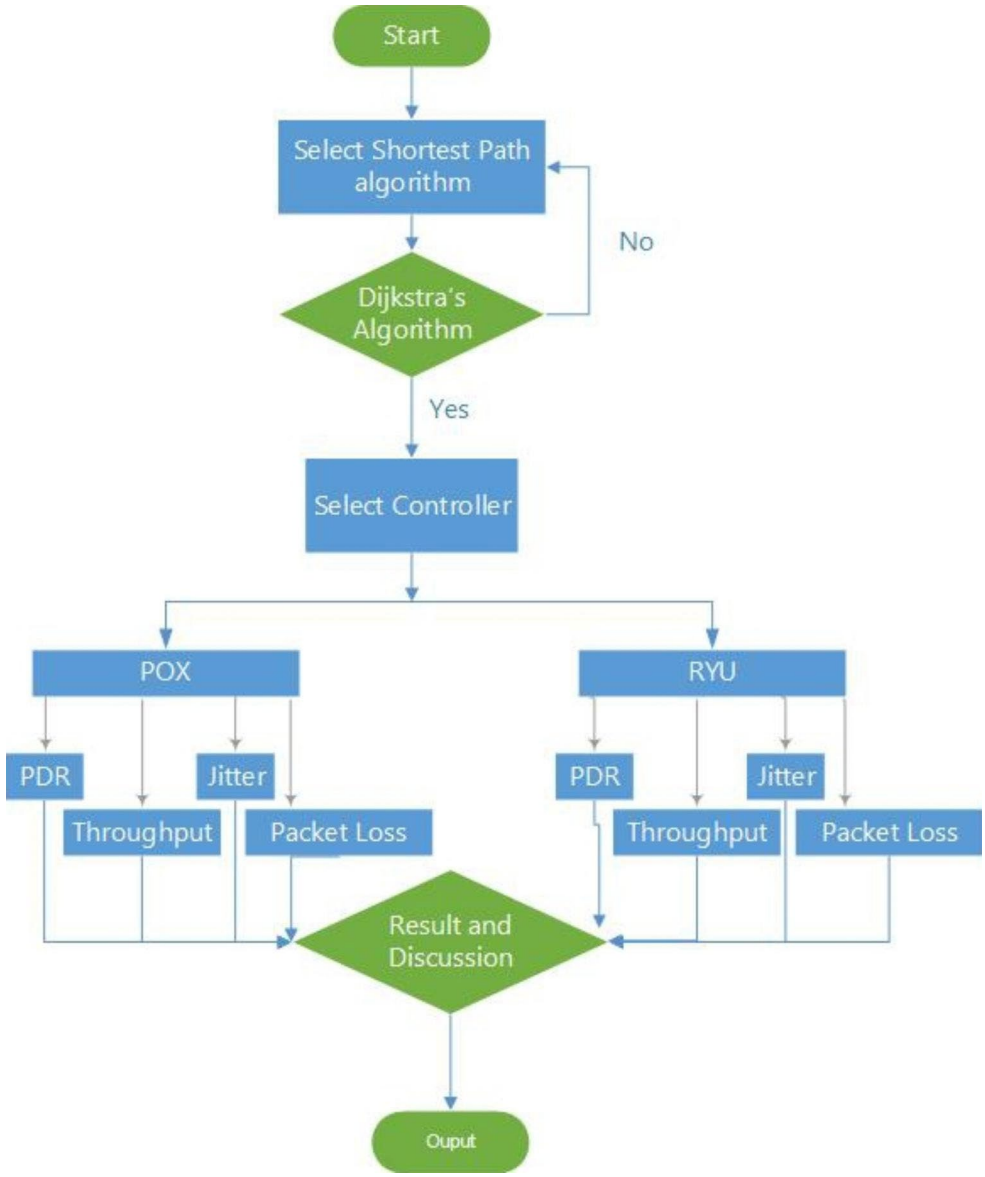

Fig. 2 Proposed Methodology

Table 2 Utilized SDN controllers

Controller	Language	Architecture	Interface	License provider	Supported platform
POX [31].	Python	Centralized	CLI, GUI	Apache	Linux, mac OS, Windows
RYU [32].	Python	Centralized	CLI, Web UI	GPL	Linux

VMware Workstation Pro is used to run the RYU and POX controllers in a virtual environment. The operating environment for the simulation is Ubuntu 18.04, which is installed in Virtual Box. A Mininet simulator is used for the network simulation, and it is capable of establishing network configurations and simulations within the scope of the simulated environment. We create a Python-based POX, RYU controller for comparing. The available throughput and packet delivery ratio of ICMP requests are measured in the context of using the shortest route algorithm. The Details about experimental setup is given below in Table 3.

4 Experimental setup

Table 3 Details of experimental setup

Category	Software used	Version
Virtual machine	VMware	14.0
Host operating system	Windows	10(64-bit)
Guest operating system	Ubuntu	18.04 (64-bit)
Emulator	Mininet	2.3.0 d4
Controllers	RYU and POX	4.34/2.0
Switch	Open v Switch	2.5.4
Southbound Comm	OpenFlow	1.3

Table 4 Throughput of the RYU controller

S. No	Time Interval (sec)	Throughput (Gbp/S)	
		Dijkstra's Algorithm	Normal Flow
1	0.0–1.0	7.16	5.01
2	1.0–2.0	7.2	5.56
3	2.0–3.0	7.14	5.56
4	3.0–4.0	7.21	5.3
5	4.0–5.0	7.16	4.62
6	5.0–6.0	7.16	4.76
7	6.0–7.0	6.81	4.39
8	7.0–8.0	7.34	4.69
9	8.0–9.0	7.23	4.73
10	9.0–10.0	7.25	5.17
Average Throughput		7.166	4.979

5 Results and discussion

This section represents the simulation results carried out via the two utilized SDN controllers using the Dijkstra's algorithm. Multiple experiments were performed for each SDN controller in combination with the Dijkstra's algorithm. Performance of the utilized SDN controllers and Dijkstra's algorithm is measured with the help of various QoS metrics such as, throughput, packet delivery ratio, jitter, and packet loss, etc. The system specifications remains the same for all the simulation results. The following subsections illustrates the results of QoS metrics attained via the SDN controllers using the Mininet tool.

5.1 Throughput

The amount of data transferred in a given amount of time is known as throughput. By processing many bits per unit of time, the throughput can be estimated numerically. The following formula is used to compute the average throughput:

$$Throughput = (\Sigma received packet size) / time\,(Gbps) \quad (1)$$

Table 5 Throughput of the POX controller

S. No	Time Interval (sec)	Throughput (Gbp/S)	
		Dijkstra's Algorithm	Normal Flow
1	0.0–1.0	13	8.22
2	1.0–2.0	13.3	7.54
3	2.0–3.0	12.9	8.31
4	3.0–4.0	13	8.02
5	4.0–5.0	12.7	8.08
6	5.0–6.0	11.9	8.12
7	6.0–7.0	12.9	8.12
8	7.0–8.0	12.6	8.02
9	8.0–9.0	13	7.99
10	9.0–10.0	12.9	7.49
Average Throughput		12.82	7.991

5.1.1 RYU Controller Throughput

In the RYU SDN controller, network throughput is estimated using the iperf real-time method between source and destination nodes with and without Dijkstra's algorithm and different topologies, and is measured in bits per second or data packets per second. The iperf program was used to assess the controller throughput performance by creating a TCP node-to-node connection where one node acts as the client and the other as a server. Table 4 shows the experimental results attained via the RYU controller using Dijkstra's algorithm and normal flow.

From Table 4 it is quite obvious that the average throughput of the Dijkstra's algorithm using the RYU controller is way better than that of the normal flow. The Dijkstra's algorithm attained the average throughput of 7.166 while the normal flow conquered the average throughput of 4.979 as shown in Table 4.

5.1.2 POX Controller Throughput

The PDR test between POX SDN controller source and destination nodes is measured in Table 5 using Dijkstra's algorithm and normal flow.

It can be observed from Table 5, once again the Dijkstra's algorithm using the POX controller attained better results than the normal flow. The Dijkstra's algorithm using POX controller achieved the average throughput of 12.82 where the normal flow using POX controller attained the throughput of 7.991.

5.1.3 POX and RYU Controller Throughput

The RYU, POX SDN controller's throughput test between source and destination nodes is measured in Table 6. Further, the comparative results show the significance of the utilized controllers and help in finding the best controller for the throughput of the network.

Table 6 Throughput of the RYU and POX controllers

S. No	Time Interval(sec)	RYU-Throughput (Gbp/S)		POX-Throughput (Gbp/S)	
		Dijkstra's Algorithm	Normal Flow	Dijkstra's Algorithm	Normal Flow
1	0.0–1.0	7.16	5.01	13	8.22
2	1.0–2.0	7.2	5.56	13.3	7.54
3	2.0–3.0	7.14	5.56	12.9	8.31
4	3.0–4.0	7.21	5.3	13	8.02
5	4.0–5.0	7.16	4.62	12.7	8.08
6	5.0–6.0	7.16	4.76	11.9	8.12
7	6.0–7.0	6.81	4.39	12.9	8.12
8	7.0–8.0	7.34	4.69	12.6	8.02
9	8.0–9.0	7.23	4.73	13	7.99
10	9.0–10.0	7.25	5.17	12.9	7.49
Average Throughput		7.166	4.979	12.82	7.991

Table 7 RYU PDR in Dijkstra's

S. No	Time Interval (sec)	RYU PDR in Dijkstra's			
		Total Packet	Packet Lost	Received	PDR
1	0.0–1.0	89,208	1514	87,694	96.41
2	1.0–2.0	89,166	557	88,609	98.44
3	2.0–3.0	89,145	315	88,830	99.87
4	3.0–4.0	89,185	406	88,779	99.12
5	4.0–5.0	89,156	251	88,905	98.35
6	5.0–6.0	89,173	327	88,846	97.34
7	6.0–7.0	89,164	178	88,986	99.27
8	7.0–8.0	88,525	72	88,453	98.09
9	8.0–9.0	89,797	207	89,590	99.04
10	9.0–10	89,130	146	88,984	99.35
Average PDR		89164.9	397.3	88767.6	98.52

The throughput numbers RYU and POX for each controller are shown in (Gbp/s) in Table 6. Further, Table 6 shows a comparison of the average network throughput of the utilized controllers. As a result, when compared to RYU, a controller for custom and conventional network topologies, the POX controller has the highest throughput value. When a large number of switches are used to evaluate network overload, the POX controller outperforms RYU. The RYU controller, on the other hand, has the lowest throughput value.

5.2 Packet delivery ratio

Packet delivery ratio (PDR) is an extremely important feature for measuring the performance of a routing system in any network. Simulation setting is also one of the factors that help in determining the performance of the protocol. Therefore it is of great interest to choose the optimal setting for the simulation in order to attain better performance results for the protocols. The packet delivery ratio is derived by dividing the total number of data packets received by the destinations by the total number of data packets supplied from sources. High packet delivery means improved performance, so when the packet delivery ratio is high the performance will be high. In this study, the packet received ratio is calculated using the below formula.

Table 8 RYU PDR in normal flow

S. No	Time Interval (sec)	RYU PDR in Normal Flow			
		Total Packet	Packet Lost	Received	PDR
1	0.0–1.0	43,647	1566	42,081	96.41
2	1.0–2.0	20,660	321	20,339	98.44
3	2.0–3.0	38,220	47	38,173	99.87
4	3.0–4.0	41,521	365	41,156	99.12
5	4.0–5.0	39,176	646	38,530	98.35
6	5.0–6.0	36,327	965	35,362	97.34
7	6.0–7.0	43,565	315	43,250	99.27
8	7.0–8.0	43,839	834	43,005	98.09
9	8.0–9.0	45,491	434	45,057	99.04
10	9.0–10	43,320	278	43,042	99.35
Average PDR		39,576	577	38999.5	98.52

$$PDR = 100 * (Delivered) / (Total) \tag{2}$$

5.2.1 PDR of RYU Controller

The PDR of the RYU controller in Dijkstra's and normal flow is shown in Tables 7 and 8 respectively.

5.2.2 PDR of POX Controller

The PDR attained via the usage of Dijkstra's algorithm is shown in Table 9. The time interval for each iteration is kept as 1.0. An average of 766534.6 packets were selected in 10 iterations in which an average of 15173.8 packets were lost, and an average of 751360.8 packets were received successfully at the destinations. Further, the average PDR attained using Dijkstra's algorithm is 97.33% which indeed is a promising value.

5.2.3 RYU and POX Controller PDR

Table 10 depicts a comparison of the two utilized SDN controllers (RYU and POX). Further, the comparison of the controllers is carried out using both Dijkstra's algorithm and normal flow. Like the other simulations the iterations are kept similar and the time interval for each iteration is 1.0. The experimental results show that the RYU algorithm in combination with Dijkstra's algorithm outperformed the POX controller by attaining an average PDR of 99.549%.

Table 9 POX-PDR in Dijkstra's algorithm

S. No	Time		POX-PDR in Dijkstra's		
	Interval	Total Packet	Packet Lost	Received	PDR (%)
1	0.0–1.0	450,936	22,989	427,947	94.90
2	1.0–2.0	400,557	18,188	382,369	95.45
3	2.0–3.0	489,888	15,437	474,451	96.84
4	3.0–4.0	1,281,242	11,401	1,269,841	99.11
5	4.0–5.0	567,057	12,468	554,589	97.80
6	5.0–6.0	1,471,233	10,979	1,460,254	99.25
7	6.0–7.0	656,060	13,354	642,706	97.96
8	7.0–8.0	1,335,629	11,310	1,324,319	99.15
9	8.0–9.0	529,291	15,494	513,797	97.07
10	9.0–10.0	483,453	20,118	463,335	95.83
Average PDR		766534.6	15173.8	751360.8	97.33

Table 10 POX & RYU controller PDR%

S. No	Time Interval	RYU Packet Delivery Ratio (%)		POX Packet Delivery Ratio (%)	
		Dijkstra's algorithm	Normal Flow	Dijkstra's Algorithm	Normal Flow
1	0.0–1.0	98.30	94.9	94.9	92.8
2	1.0–2.0	99.37	95.45	95.45	95.19
3	2.0–3.0	99.64	96.84	96.84	98.01
4	3.0–4.0	99.54	99.11	99.11	93.67
5	4.0–5.0	99.71	97.8	97.8	95.82
6	5.0–6.0	99.63	99.25	99.25	96.01
7	6.0–7.0	99.80	97.96	97.96	98.53
8	7.0–8.0	99.91	99.15	99.15	93.34
9	8.0–9.0	99.76	97.07	97.07	98.69
10	9.0–10.0	99.83	95.83	95.83	97.93
Average PDR		99.549	98.528	97.336	95.99

Table 11 Jitter values of RYU Controller

S. No	Time Interval(sec)	RYU-Jitter in Dijkstra Algorithm		RYU-Jitter in Normal Flow	
		Total Packet	Jitter(ms)	Total Packet	Jitter(ms)
1	0.0–1.0	89,208	0.001	43,647	0.065
2	1.0–2.0	89,166	0.002	20,660	0.015
3	2.0–3.0	89,145	0.002	38,220	0.014
4	3.0–4.0	89,185	0.001	41,521	0.019
5	4.0–5.0	89,156	0.002	39,176	0.021
6	5.0–6.0	89,173	0.001	36,327	0.025
7	6.0–7.0	89,164	0.001	43,565	0.019
8	7.0–8.0	88,525	0.001	43,839	0.011
9	8.0–9.0	89,797	0.003	45,491	0.026
10	9.0–10.0	89,130	0.001	43,320	0.013
Average Jitter		89164.9	0.0015	39576.6	0.0228

5.3 Jitter

In the previous experiments, jitter was defined as the variation in packet delay, or the time delay between when a packet is transmitted and when it is received. Jitter is measured by making UDP connections between POX and RYU controller for various numbers of times in standard and custom Mininet topologies.

Table 12 POX Controller Jitter

S.No	Time Interval	POX-Jitter in Dijkstra		POX-Jitter in Normal Flow	
		Total Packet	Jitter(ms)	Total Packet	Jitter(ms)
1	0.0–1.0	450,936	0.003	23,981	0.008
2	1.0–2.0	400,557	0.004	26,629	0.062
3	2.0–3.0	489,888	0.005	85,326	0.017
4	3.0–4.0	1,281,242	0.014	75,525	0.02
5	4.0–5.0	567,057	0.003	82,945	0.021
6	5.0–6.0	1,471,233	0.003	85,998	0.012
7	6.0–7.0	656,060	0.002	82,485	0.017
8	7.0–8.0	1,335,629	0.003	84,643	0.026
9	8.0–9.0	529,291	0.005	84,910	0.015
10	9.0–10.0	483,453	0.002	82,493	0.017
Average Jitter		766534.6	0.0044	71493.5	0.0215

5.3.1 Jitter of the RYU controller

The jitter value calculated for different packets of the RYU controller in Dijkstra's algorithm and normal flow is shown below in Table 11.

5.3.2 Jitter of the POX controller

Jitter for different packets of the POX controller in Dijkstra's algorithm and normal flow is shown below in Table 12.

5.3.3 Jitter of the RYU and POX Controllers

Table 13 demonstrates the jitter values and a comparison of the two utilized SDN controllers. From Table 13, it is quite obvious that the Jitter performance of the RYU controller is better than the POX controller.

Table 13 Jitter of the RYU and POX Controllers

S. No	Time Interval(sec)	RYU Jitter (ms)		POX-Jitter (ms)	
		Dijkstra's Algorithm	Normal Flow	Dijkstra's Algorithm	Normal Flow
1	0.0–1.0	0.001	0.065	0.003	0.008
2	1.0–2.0	0.002	0.015	0.004	0.062
3	2.0–3.0	0.002	0.014	0.005	0.017
4	3.0–4.0	0.001	0.019	0.014	0.02
5	4.0–5.0	0.002	0.021	0.003	0.021
6	5.0–6.0	0.001	0.025	0.003	0.012
7	6.0–7.0	0.001	0.019	0.002	0.017
8	7.0–8.0	0.001	0.011	0.003	0.026
9	8.0–9.0	0.003	0.026	0.005	0.015
10	9.0–10.0	0.001	0.013	0.002	0.017
Average Jitter		0.0015	0.0228	0.0044	0.0215

5.4 Packet loss

Data is transferred and retrieved in little units known as packets in every network system. Data packets that do not reach at their destination after being transmitted across a computer network are referred to as packet loss. Packet loss also refers to the number of packets lost or deleted during their transit through a computer network.

5.4.1 RYU Controller Packet loss

The packet loss ratio variability of the RYU SDN controller in Dijkstra's algorithm and normal flow (without Dijkstra's), the measuring of packet loss by making UDP connections between client and server of RYU for different number of times in custom Mininet topology, is shown in Table 14.

5.4.2 Packet loss of the POX Controller

The packet loss ratio variability of the POX SDN controller in Dijkstra's algorithm and normal flow is shown in the Table 15. Packet loss is measured by creating UDP connections between POX's client and server for various numbers of times in a custom mininet topology.

Table 14 Packet Loss of the RYU Controller

S. No	Time Interval(sec)	RYU Packet-Lost in Dijkstra's			RYU Packet-Lost in Normal Flow		
		Total Packet	Packet Lost	Packet Lost %	Total Packet	Packet Lost	Packet Lost%
1	0.0–1.0	89,208	1514	1.69	43,647	1566	3.58
2	1.0–2.0	89,166	557	0.62	20,660	321	1.55
3	2.0–3.0	89,145	315	0.35	38,220	47	0.12
4	3.0–4.0	89,185	406	0.45	41,521	365	0.87
5	4.0–5.0	89,156	251	0.28	39,176	646	1.64
6	5.0–6.0	89,173	327	0.36	36,327	965	2.65
7	6.0–7.0	89,164	178	0.19	43,565	315	0.72
8	7.0–8.0	88,525	72	0.08	43,839	834	1.90
9	8.0–9.0	89,797	207	0.23	45,491	434	0.95
10	9.0–10.0	89,130	146	0.16	43,320	278	0.64
Average Packet Lost		89164.9	397.3	0.441	39576.6	577.1	1.462

Table 15 POX Controller Packet Loss

S. No	Time Interval	POX Packet Lost in Dijkstra's			POX Packet Lost in Normal Flow		
		Total Packet	Packet lost	Packet Lost (%)	Total Packet	Packet Lost	Packet Lost (%)
1	0.0–1.0	450,936	22,989	5.09	23,981	1726	7.19
2	1.0–2.0	400,557	18,188	4.54	26,629	1280	4.80
3	2.0–3.0	489,888	15,437	3.15	85,326	1703	1.99
4	3.0–4.0	1,281,242	11,401	0.88	75,525	4774	6.32
5	4.0–5.0	567,057	12,468	2.19	82,945	3460	4.17
6	5.0–6.0	1,471,233	10,979	0.74	85,998	3430	3.98
7	6.0–7.0	656,060	13,354	2.03	82,485	1210	1.46
8	7.0–8.0	1,335,629	11,310	0.84	84,643	5630	6.65
9	8.0–9.0	529,291	15,494	2.92	84,910	1105	1.30
10	9.0–10.0	483,453	20,118	4.16	82,493	1700	2.06
Average		766534.6	15173.8	2.654	71493.5	2601.8	3.992

Table 16 Packet Loss of RYU vs. POX Controllers

S. No	Time Interval(sec)	RYU Packet Lost (%)		POX Packet Lost (%)	
		Dijkstra's Algorithm	Normal Flow	Dijkstra's Algorithm	Normal Flow
1	0.0–1.0	1.69	3.58	5.09	7.19
2	1.0–2.0	0.62	1.55	4.54	4.80
3	2.0–3.0	0.35	0.12	3.15	1.99
4	3.0–4.0	0.45	0.87	0.88	6.32
5	4.0–5.0	0.28	1.64	2.19	4.17
6	5.0–6.0	0.36	2.65	0.74	3.98
7	6.0–7.0	0.19	0.72	2.03	1.46
8	7.0–8.0	0.08	1.90	0.84	6.65
9	8.0–9.0	0.23	0.95	2.92	1.30
10	9.0–10.0	0.16	0.64	4.16	2.06
Average Packet Lost		0.441	1.462	2.654	3.992

5.4.3 RYU and POX Controller Packet loss

In this subsection the simulation results of the RYU and POX SDN controllers are compared. Table 16 compares the packet loss of the RYU and POX controllers.

Table 16 shows that, the RYU with Dijkstra's algorithm performed really well and has the smallest packet loss ratio i.e. 0.441. The RYU also performed well with normal flow and has 1.462 packet loss ratio. POX with and without Dijkstra's algorithm performed poorly has the packet loss ratio of 2.654 and 3.992 respectively for Dijkstra and without Dijkstra's algorithm.

6 Conclusions

The traditional networks lack scalability when scaling up or down since they rely solely on hardware components such as switches and routers. A new technology known as SDN can help to tackle this problem. SDN allows network managers to separate the control plane from the data plane and logically consolidate control operations in a software controller. SDN can aid in the management of dynamic traffic while also safeguarding the network's stability, privacy, and quality of service limits. Many traffic management controllers are featured in the SDN or programmable networks. Each one is distinct from the others according to its level of complexity and the level of assistance supplied by its developer. In this study, Dijkstra's shortest path algorithm is used along with the SDN controllers (RYU and POX), allowing for better SDN behavior in terms of various QoS metrics and finding the optimum and shortest path for the network packets. The Mininet emulator is used for carrying out the simulation results. Performance of the two utilized controllers is measured in terms of various QoS metrics such as throughput, packet delivery ratio, jitter, and packet loss on a specific network architecture under a variety of workloads. The comparison of the POX and RYU controllers reveals that the POX controller outperforms the RYU controller in terms of throughput. The RYU controller performs better in terms of packet delivery ratio, jitter, and packet loss. We have a plan to utilize various controllers in the future to verify and enhance the performance, such as Open daylight, Trema, and Floodlight, and to use other parameters like end-to-end delay, packet loss, and bitrate.

References

1. Akbar, A., Ibrar, M., Jan, M. A., Bashir, A. K., & Wang, L. (2020). SDN-enabled adaptive and reliable communication in IoT-fog environment using machine learning and multiobjective optimization. *IEEE Internet of Things Journal*, *8*(5), 3057–3065
2. Zhu, L., Karim, M. M., Sharif, K., Xu, C., Li, F., Du, X., & Guizani, M. (2020). "SDN controllers: A comprehensive analysis and performance evaluation study,". *ACM Computing Surveys (CSUR)*, *53*, 1–40
3. Trois, C., Del Fabro, M. D., de Bona, L. C., & Martinello, M. (2016). "A survey on SDN programming languages: Toward a taxonomy,". *IEEE Communications Surveys & Tutorials*, *18*, 2687–2712
4. Sinha, Y., & Haribabu, K. (2017). A survey: Hybrid sdn,. *Journal of Network and Computer Applications*, *100*, 35–55
5. Benzekki, K., Fergougui, A. E., & Elbelrhiti Elalaoui, A. (2016). "Software-defined networking (SDN): a survey,". *Security and Communication Networks*, *9*, 5803–5833
6. Bannour, F., Souihi, S., & Mellouk, A. (2017). Distributed SDN control: Survey, taxonomy, and challenges,. *IEEE Communications Surveys & Tutorials*, *20*, 333–354
7. Karakus, M., & Durresi, A. (2017). "A survey: Control plane scalability issues and approaches in software-defined networking (SDN)," *Computer Networks*, vol. 112, pp. 279–293,
8. Han, T., Jan, S. R. U., Tan, Z., Usman, M., Jan, M. A., Khan, R., & Xu, Y. (2020). A comprehensive survey of security threats and their mitigation techniques for next-generation SDN controllers. *Concurrency and Computation: Practice and Experience*, *32*(16), e5300
9. Campbell, A. T., Katzela, I., Miki, K., & Vicente, J. (1999). "Open signaling for ATM, internet and mobile networks (OPENSIG'98)," ACM SIGCOMM Computer Communication Review, vol. 29, pp. 97–108,
10. Nunes, B. A. A., Mendonca, M., Nguyen, X. N., Obraczka, K., & Turletti, T. (2014). A survey of software-defined networking: Past, present, and future of programmable networks,. *IEEE Communications surveys & tutorials*, *16*, 1617–1634
11. Al-Tam, F., & Correia, N. (2019). Fractional switch migration in multi-controller software-defined networking,. *Computer Networks*, *157*, 1–10
12. Niculescu, D. (2004). "Survey of active network research," Retrieved on Jul, vol. 22, p. 1999
13. Van der Merwe, J. E., Rooney, S., Leslie, L., & Crosby, S. (1998). The Tempest-a practical framework for network programmability,. *IEEE network*, *12*, 20–28
14. Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., et al. (2008). "NOX: towards an operating system for networks," ACM SIGCOMM Computer Communication Review, vol. 38, pp. 105–110,

15. Tran, H. M., Tumar, I., & Schönwälder, J. (2009). "NETCONF interoperability testing," in IFIP International Conference on Autonomous Infrastructure, Management and Security, pp. 83–94,
16. Azodolmolky, S. (2013). *Software defined networking with OpenFlow*. Packt Publishing Ltd
17. Ranjan, P., Pande, P., Oswal, R., Qurani, Z., & Bedi, R. (2014). "A survey of past present and future of software defined networking,"International Journal of Avance Research in Computer Science and Management Studies, vol. 2,
18. Hande, Y. S., & Akkalakshmi, M. (2015). "A Study on Software Defined Networking,"International Journal of Innovative Research in Computer and Communication Engineering, vol. 3,
19. Sherwood, R., & Yap, K. (2011). "Cbench controller benchmarker," Last accessed, Nov,
20. Goransson, P., Black, C., & Culver, T. (2016). *Software defined networks: a comprehensive approach*. Morgan Kaufmann
21. Abdullah, M. Z., Al-Awad, N. A., & Hussein, F. W. (2018). "Performance Evaluation and Comparison of Software Defined Networks Controllers,". *International Journal of Scientific Engineering and Science*, *2*, 45–50
22. Wang, C., Zhang, G., Xu, H., & Chen, H. (2016). "An ACO-based link load- balancing algorithm in SDN," in *7th International Conference on Cloud Computing and Big Data (CCBD)*, pp. 214–218,
23. Kaur, K., Kaur, S., & Gupta, V. (2016). "Least time based weighted load balancing using software defined networking," in *International Conference on Advances in Computing and Data Sciences*, pp. 309–314,
25. Duque, J. P., Beltrán, D. D., & Leguizamón, G. P. (2018). "OpenDaylight vs. Floodlight: Comparative Analysis of a Load Balancing Algorithm for Software Defined Networking,". *International Journal of Communication Networks and Information Security*, *10*, 348–357
26. Joshi, & Gupta, D. (2019). "A Comparative Study on Load Balancing Algorithms in Software Defined Networking," in *International Conference on Ubiquitous Communications and Network Computing*, pp. 142–150,
28. Sajid, A. S., Niloy, S. F. N., Hossain, K., & Rahman, T. (2018). *"Comprehensive evaluation of shortest path algorithms and highest bottleneck bandwidth algorithm in software defined networks,"*. BRAC University
29. Jarraya, Y., Madi, T., & Debbabi, M. (2014). "A survey and a layered taxonomy of software-defined networking,". *IEEE communications surveys & tutorials*, *16*, 1955–1980
30. Jain, S. (2018). "Performance Evaluation of Multi Hop Routing Using Software Defined Network," in *Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, pp. 1–5, 2018
31. Satre, S. M., Patil, N. S., Khot, S. V., & Saroj, A. A. (2020). "Network Performance Evaluation in Software-Defined Networking," presented at the International Conference on Information and Communication Technology for Intelligent Systems,
32. Ryu, S., "Framework Community,"Ryu Controller."," ed
33. Ryu, S., "Framework (sd)," Récupéré sur https://osrg.github.io/ryu
35. Sufiev, H., & Haddad, Y. (2016). "A dynamic load balancing architecture for SDN," in *IEEE International Conference on the Science of Electrical Engineering (*ICSEE), pp. 1–3, 2016

Yingjun Zhang was born in Xinjiang in 1984. He received the M.S. degree in control theory and control engineering from Guangxi Technology University in 2011. He received his the Ph.D. in Air Force Engineering University. His research interests are in fault diagnosis control, iterative learning control and deep learning.

Mengji Chen was born in Guangxi in 1991, teacher of Hechi University, is engaged in electromechanical integration design and electromechanical control research. He has published 8 papers and applied for 10 Chinese utility model patents.