Int. J. Advanced Networking and Applications
Volume: 17, Issue: 03, Pages : 6954-6959 (2025) ISSN: 0975-0290
https://doi.org/10.35444/IJANA.2025.17308

Performance Evaluation of SDN Controllers for Graph Detection Time

Tareg Abubaker Abulifa
Department of Electronic Engineering, College of Industrial Technology, Misurata-Libya
Email: Tareg_Abulifa@cit.edu.ly
Salem Omar Sati
Faculty of Information Technology, Misurata University, Misurata-Libya
Email: salem.sati@it.misuratau.edu.ly

ABSTRACT

Software Defined Networks (SDNs) propose the decoupling of the control plane from the forwarding plane, enabling centralized management through OpenFlow protocol [1]. Current performance evaluation tools for SDN architectures primarily focus on basic attributes like throughput and latency, overlooking the critical metric of Graph Detection time. This study bridges this gap by introducing a novel Python-based monitoring tool that evaluates Graph Detection time for Java-based SDN controllers. The tool measures detection speed across various network sizes using REST northbound interfaces with dedicated control channels. Our comprehensive evaluation of OpenDaylight (ODL) and Open Network Operating System (ONOS) controllers demonstrates that ODL consistently outperforms ONOS in Graph Detection time across all tested topologies [2]. These findings provide crucial insights for network designers implementing time-sensitive applications in large-scale SDN environments.

Keywords - **Graph Detection Time, SDN Controllers, OpenDaylight, ONOS, Performance Evaluation, Northbound API.**

Date of Submission: August 08, 2025 Date of Acceptance: September 28, 2025

I. INTRODUCTION

Software Defined Networking (SDN) represents a paradigm shift in network architecture by centralizing control logic within dedicated controller components [3]. Unlike traditional networks, where control and forwarding functions are tightly coupled within individual devices, SDN introduces a clean separation of concerns: the data plane, which is responsible for high-speed packet forwarding, and the control plane, which makes global routing and policy decisions [4]. This architectural transformation enables unprecedented programmability, simplified management, and centralized visibility across the entire network infrastructure.

Among the various southbound protocols that enable this separation, OpenFlow has emerged as the de facto standard for controller–switch communication, providing a flow-based abstraction for packet handling [5]. Through OpenFlow, controllers install forwarding rules directly into switch flow tables, enabling dynamic reconfiguration of network behavior in response to application requirements or topology changes.

The SDN controller serves as the network's "operating system", maintaining a global view of the topology, enforcing policies, and exposing programmable interfaces for applications. Prominent controllers such as NOX**,** POX, RYU, ONOS, OpenDaylight (ODL), and Floodlight embody different design philosophies and implementation choices. For instance, lightweight controllers like POX are suited for teaching and prototyping, while distributed controllers such as ONOS target carrier-grade deployments. Similarly, ODL emphasizes modularity and interoperability through its OSGi-based architecture. These variations result in distinct performance characteristics in terms of scalability, responsiveness, and reliability [6].

II. RELATER WORKS

Prior research has extensively evaluated SDN controllers through various performance lenses, though Graph Detection Time remains relatively unexplored [7]. Several studies have compared controller performance metrics such as throughput, latency, and scalability under different network conditions [8]. Research by Darianian et al. evaluated ONOS and OpenDaylight using the Cbench tool, finding ONOS superior in throughput and latency within virtualized environments [9].

Another study by Sati et al. examined how increasing OpenFlow switches impacts ODL and ONOS performance, noting ONOS maintained better metrics at scale[2]. Comparative analyses of POX versus Floodlight controllers by Bholebawa and Dalal revealed Flood light's advantages in latency and throughput [10,11], while RYU demonstrated superior performance over POX in throughput, delay, and overhead metrics [12].

Beyond controller-level benchmarking, other works have focused on the foundational aspects of SDN and its emulation environments. Pooja and Sood [13] highlighted the architectural distinctions between traditional networks and SDN, emphasizing how centralized control, programmability, and open interfaces enable more dynamic and scalable designs. Their study further presented Mininet as a cost-effective tool to emulate and analyze SDN behavior, demonstrating its utility for rapid

Int. J. Advanced Networking and Applications
Volume: 17, Issue: 03, Pages : 6954-6959 (2025) ISSN: 0975-0290
https://doi.org/10.35444/IJANA.2025.17308

prototyping and academic experimentation. Similarly, Lantz et al. [14] described Mininet's role in enabling large-scale network emulation on commodity hardware, making it a widely adopted platform for both teaching and research. Additional works have also investigated SDN programmability through languages such as Frenetic, Pyretic, and Procera [3], which provide abstraction layers for developing advanced network policies.

Recent investigations into containerized SDN control planes and Packet-In filtering mechanisms highlight evolving architectural approaches [15, 16]. However, comprehensive evaluation of topology discovery efficiency remains limited, creating a significant research gap our study addresses.

III. Software Defined Networking Fundamentals

Software Defined Networking (SDN) fundamentally transforms network management by decoupling control logic from forwarding hardware [14]. This architectural separation creates two distinct operational planes: the data plane, which handles packet forwarding at line speed, and the control plane, which makes intelligent routing decisions through centralized controllers [17]. These controllers communicate with network switches through southbound APIs like OpenFlow [18], while exposing northbound REST APIs for application integration. This layered abstraction simplifies network programming by allowing developers to create applications without deep knowledge of underlying hardware specifics [19]. The controller's centralized intelligence enables dynamic traffic engineering, policy enforcement, and network virtualization. REST APIs following Representational State Transfer principles provide standardized access to network state information using HTTP methods [20].

In traditional networks, both control and data functions are tightly coupled within individual devices such as routers and switches, making policy updates time-consuming and error-prone. By contrast, SDN centralizes the control logic, significantly reducing complexity while improving flexibility and scalability [14]. This decoupling enables network operators to respond rapidly to evolving requirements such as mobile device proliferation, security demands, and big data traffic patterns. Moreover, SDN fosters a programmable paradigm where external applications can dynamically reconfigure flows, optimize routing, and enforce security rules in near real time [17].

The architecture also introduces distinct northbound and southbound interfaces. The southbound interface, with OpenFlow as the de facto standard, governs communication between controllers and forwarding devices. Northbound interfaces, on the other hand, allow high-level applications to interact with the controller for tasks such as monitoring, load balancing, and anomaly detection.

Another critical feature of SDN is its support for network virtualization. Controllers abstract the underlying physical infrastructure into logical topologies that can be independently programmed and managed. This abstraction makes it possible to run multiple virtual networks over shared hardware resources, providing efficiency and isolation in cloud and data center environments [17].

Finally, SDN's programmability and centralized visibility provide fertile ground for research in performance optimization. Prior work has shown that SDN can enable rapid prototyping of new protocols, flexible traffic engineering strategies, and enhanced security mechanisms [14, 17]. However, while controller throughput, latency, and scalability have been widely studied, aspects such as graph detection efficiency and topology discovery latency remain underexplored issues that are critical for large-scale and dynamic environments. Addressing these gaps requires not only benchmarking controller performance but also examining the efficiency of network state synchronization and topology inference mechanisms.

IV. Control Plane Architectures

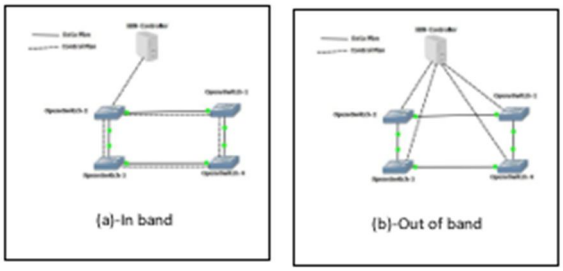


Figure 1: SDN control plane architectures: (a) In-band control, (b) Out-of-band control

Control plane implementation in SDN networks follows either in-band or out-of-band configurations, each with distinct operational characteristics [18]. In-band control planes utilize the same network infrastructure for both control messages and data traffic, simplifying deployment but introducing potential performance bottlenecks and security vulnerabilities [19]. This shared pathway creates resource contention where control and data traffic compete for bandwidth, potentially degrading critical control signals during congestion. Moreover, attackers targeting the data plane can potentially disrupt controller communications, making in-band designs less resilient to denial-of-service (DoS) scenarios [14].

Conversely, out-of-band control employs dedicated channels separating control traffic from data flows [21]. This isolation provides enhanced security through physical segmentation and predictable performance unaffected by data plane congestion. Control messages maintain consistent latency characteristics essential for real-time network operations, while the independence of channels facilitates rapid fault isolation and recovery. Although out-of-band deployment requires additional infrastructure such as separate interfaces or links, this trade-off is often justified in large-scale data center and carrier-grade environments where controller responsiveness is critical [17].

Prior research has emphasized that the choice of control plane architecture directly influences overall SDN

performance. Studies leveraging Mininet have demonstrated how in-band setups can distort experimental results by introducing unintended packet delays or losses when data plane traffic intensifies [14, 20]. In contrast, out-of-band models allow more accurate evaluation of controller behavior by eliminating interference effects, making them the preferred choice for benchmarking tasks such as throughput, flow setup latency, and topology discovery.

Our experimental framework adopts the out-of-band control approach, following RFC 8456 recommendations, to eliminate data plane interference and accurately measure controller performance [19]. This design decision ensures that the evaluation of graph detection time reflects controller processing efficiency rather than congestion-induced anomalies in the shared channel. Furthermore, isolating control traffic strengthens the reliability of comparative analysis across different SDN controllers, particularly when studying scalability in large or dynamic topologies.

V. OpenFlow Protocol Analysis

The OpenFlow protocol serves as the foundational communication standard between SDN controllers and network switches, originally developed at Stanford University in 2008 [3]. By decoupling the control and forwarding planes, OpenFlow allows centralized controllers to install forwarding rules in network devices through programmable flow tables, fundamentally shifting network management from manual configuration to dynamic, software-driven control [18]. This programmability has positioned OpenFlow as the de facto southbound API for SDN, widely supported by both open-source controllers such as POX, Floodlight, RYU, and ONOS, and by commercial vendor hardware [14].

OpenFlow has evolved through multiple versions, with version 1.3 used in this study introducing significant enhancements over earlier iterations [4]. Among these are multiple flow tables, which enable more sophisticated pipeline processing; group tables, which provide abstractions for multipath forwarding, fast reroute, and load balancing; and extended match fields, accommodating VLAN tags, MPLS labels, and tunneling protocols. These extensions allow controllers to define more granular and scalable forwarding policies, making OpenFlow 1.3 suitable for large-scale and heterogeneous environments [17].

For the purposes of this study, the role of OpenFlow is particularly significant in evaluating graph detection time. Our measurement framework leverages Statistics messages to determine when topology discovery has completed by tracking flow and port state synchronization between switches and the controller. The efficiency with which controllers process Features, Packet-In, and Statistics messages directly affects the latency of network graph construction. Thus, analyzing OpenFlow 1.3 behavior not only provides insights into flow-level performance but also exposes underlying factors influencing topology inference and detection efficiency.

VI. Controller Architectures

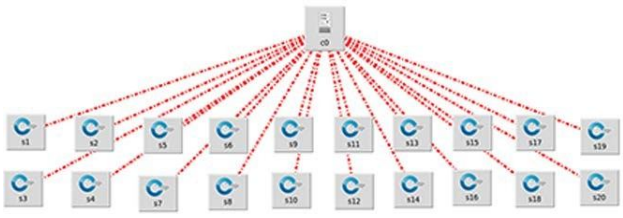
Two prominent Java-based SDN controllers were evaluated in this study: OpenDaylight (ODL) and the Open Network Operating System (ONOS) [7, 8].

OpenDaylight, developed under the Linux Foundation, is one of the earliest large-scale community-driven SDN controller projects. It features a modular architecture based on the OSGi framework, where services can be dynamically loaded or removed at runtime. Its Karaf-based customization enables selective feature activation, and it supports multiple southbound protocols including OpenFlow (v1.0–1.3)**,** NETCONF, and BGP-LS [8]. ODL emphasizes interoperability and extensibility, with its RESTCONF API (/restconf/operational/network-topology) providing standardized access to topology and network state. This flexibility has made it widely adopted in both research and industry, although prior studies have noted performance trade-offs such as increased latency under high flow setup rates [9].

ONOS, by contrast, was designed with a distributed core architecture, explicitly targeting carrier-grade and multi-domain SDN deployments [7]. Its microservices-based model provides fault tolerance and scalability by allowing multiple controller instances to share state across a cluster. ONOS supports OpenFlow, OVSDB, and NETCONF through provider applications, implemented as OSGi bundles that handle device-specific communication. Its management ecosystem includes a web-based GUI, a CLI, and REST APIs (e.g., /onos/v1/devices with onos/rocks credentials), enabling integration with higher-level orchestration systems. Studies have shown ONOS to exhibit superior throughput and latency performance compared to ODL, particularly as the number of switches and flows scale [2].

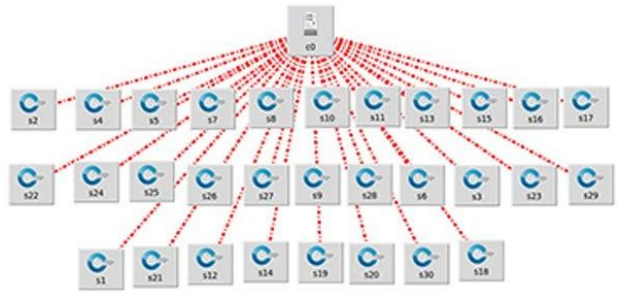
Both controllers were evaluated in their default configurations using dedicated out-of-band control channels to ensure measurement accuracy. By isolating control traffic from data-plane flows, the experiments ensure that observed differences in graph detection time stem from architectural and implementation factors rather than interference effects.

VII. Scalability Topologies

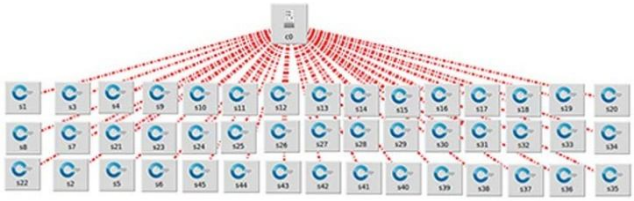


(a) 20 switches (small)

Int. J. Advanced Networking and Applications
Volume: 17, Issue: 03, Pages : 6954-6959 (2025) ISSN: 0975-0290
https://doi.org/10.35444/IJANA.2025.17308



(b) 30 switches (medium)



(c) 45 switches (large)

Figure 2: Scalability topologies with dedicated control channels

Scalability topologies were designed to evaluate controller performance across increasing network sizes while maintaining dedicated control channels [19]. The experimental framework comprised three distinct topologies containing 20, 30, and 45 virtual switches respectively, each directly connected only to the SDN controller via out-of-band channels [21]. This configuration isolates control plane traffic from data plane interference, enabling accurate measurement of controller responsiveness. Following RFC 8456 benchmarking methodology, each topology maintained dedicated OpenFlow channels equivalent to the number of switches [19]. Crucially, no inter-switch data plane connections existed, ensuring control traffic traveled exclusively through dedicated pathways. The topology design facilitates precise evaluation of controller performance under scaling conditions by eliminating confounding variables from data traffic. By connecting virtual switches exclusively to controllers, the framework enables focused assessment of control plane efficiency during topology discovery. This approach minimizes interference between control and data planes while simplifying troubleshooting during experimentation [14].

VIII. GRAPH DETECTION MEASUREMENT TOOL

Our Python-based measurement tool addresses significant limitations in existing SDN evaluation utilities like Cbench, which lacks Graph Detection Time assessment and only supports obsolete OpenFlow versions [9]. The developed solution implements a comprehensive approach to topology discovery timing using REST northbound interfaces [20]. The measurement algorithm initiates timing upon first switch connection establishment, continuously polling the controller's device list through REST API calls. Detection completion occurs when all switches appear as "available" in the controller's topology view, with precise millisecond timing capturing the discovery duration [16]. The implementation consists of specialized classes for each controller platform, leveraging their unique REST API characteristics while maintaining a consistent measurement methodology. This approach provides accurate, vendor-agnostic measurement of topology discovery efficiency across controller platforms while supporting modern OpenFlow 1.3 implementations [18].

IX. IMPLEMENTATION DETAILS

The Graph Detection Time measurement tool implements specialized Python classes for each controller platform, with complete code available in the supplementary figures directory [20]. The ODL implementation shown in Figure 3 utilizes the topology inventory endpoint with admin credentials in the request header. This class establishes HTTP connections using httplib2 with credential authentication [17]. The get switches number method queries the operational nodes endpoint and parses the JSON response to count connected switches. The get Convergence Time method implements the core timing logic, initiating measurement when no switches are detected and terminating when all target switches appear in the topology [16].

```
class ODL:

    def __init__(self):
        self.baseUrl = ('http://IP:Port/restconf')
        self.h = httplib2.Http(".cache")
        self.h.add_credentials('admin', 'admin')

    def get_switches_number(self):
     try:
      url = self.baseUrl + '/operational/opendaylight-inventory:nodes'
      logging.debug('url %s', url)
      _,content = self.h.request(url, "GET")
      allContent = json.loads(content)
      node_list=allContent['nodes']
      if node_list == {}:    ### if topology is empty return 0
       return 0
      return (len(node_list['node'])) ### return number of items in nodes (switches) list
     except:
       return 0


    def getConvergenceTime(self):
     try :
      nodes = self.get_switches_number()
      max_nodes = 20

      if(nodes != 0):  ### first request must return 0 switches to store start time
        return 0

      while(nodes < max_nodes):

        if(nodes == 0): ### storing start time
            self.start_time = int(time.time() * 1000)

        if(nodes >= max_nodes): ### storing endtime
          self.end_time = int(time.time() * 1000)

        nodes = self.get_switches_number()

      return int(self.end_time - self.start_time)
     except Exception as e:
      return 0
```

Figure 3: OpenDaylight Graph Detection Time implementation

The ONOS implementation presented in Figure 4 follows a similar pattern but uses different API endpoints and authentication credentials [7]. The get switches number method queries the /devices endpoint and iterates through the response to count available switches. The implementation specifically checks the 'available'

Int. J. Advanced Networking and Applications
Volume: 17, Issue: 03, Pages : 6954-6959 (2025) ISSN: 0975-0290
https://doi.org/10.35444/IJANA.2025.17308

attribute to confirm active connections [17]. Both classes employ identical timing methodology but adapt to controller-specific API structures. The implementations include robust error handling through try-except blocks and logging for operational diagnostics [15]. The measurement precision reaches millisecond resolution using Python's time module, ensuring accurate performance comparisons between controllers [19].

Key implementation features include authentication handling through HTTP headers, JSON response parsing for topology data, and precise timing mechanisms [20]. The ODL implementation processes node lists from the inventory endpoint, while the ONOS version checks device availability status [8]. Both implementations follow the same algorithmic workflow: initialize timing at first switch detection, continuously poll controller state, and terminate timing when all target switches become available [16]. The code structure enables easy adaptation for evaluating additional controllers by implementing similar methods for their respective REST APIs. These implementations demonstrate practical application of northbound interfaces for performance monitoring in SDN environments [19].

```
class ONOS:

    def __init__(self):
        self.baseUrl = ('http://IP:Port/onos/v1')
        self.h = httplib2.Http(".cache")
        self.h.add_credentials('onos', 'rocks')

    def get_switches_number(self):
      url = self.baseUrl + '/devices'
      logging.debug('url %s', url)
      _,content = self.h.request(url, "GET")
      allContent = json.loads(content)
      node_list=allContent['devices']

      counter = 0
      for switch in node_list:
        if(switch['available'] == True):
          counter = counter+1
      return (counter)


    def getConvergenceTime(self):
     try :
      nodes = self.get_switches_number()
      max_nodes = 20

      if(nodes != 0):  ### first request must return 0 switches to store start time
        return 0

      while(nodes < max_nodes):

        if(nodes == 0): ### storing start time
            self.start_time = int(time.time() * 1000)

        if(nodes >= max_nodes): ### storing endtime
          self.end_time = int(time.time() * 1000)

        nodes = self.get_switches_number()

      return int(self.end_time - self.start_time)
     except Exception as e:
      return 0
```

Figure 4: ONOS Graph Detection Time implementation

X. Experimental Results

Experimental results demonstrate significant performance differences between controllers across all tested topologies [2]. OpenDaylight consistently outperformed ONOS in Graph Detection Time, with the performance gap widening as network size increased [21]. For the 20-switch topology, ODL completed discovery in 0.5 seconds compared to ONOS's 0.7 seconds, representing a 40% advantage. At 30 switches, ODL maintained its lead with 0.6 seconds versus ONOS's 1.0 seconds, a 67% improvement [9]. The largest topology revealed the most substantial difference, where ODL detected all 45 switches in 1.0 seconds while ONOS required 1.7 seconds, showing 70% faster performance [2]. Scaling analysis revealed important trends: increasing from 20 to 30 switches caused ODL's detection time to rise by 20% versus ONOS's 80% increase [12]. Similarly, expanding from 30 to 45 switches resulted in a 30% increase for ODL compared to 60% for ONOS [17]. These findings indicate ODL's architecture better accommodates network scaling while maintaining efficient topology discovery operations [8]. Figure 5 shows Graph Detection Time Comparison of ODL and ONOS Controllers Across Topologies

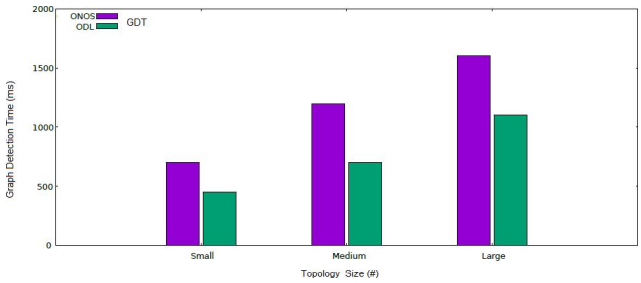


Figure 5: Graph Detection Time Comparison of ODL and ONOS Controllers Across Topologies

XI. Conclusion and Future Work

This study intreduced a novel methodology for evaluating Graph Detection Time in SDN controllers using a purpose-built Python measurement tool. Unlike existing benchmarking utilities, the proposed framework leverages REST northbound APIs and supports OpenFlow 1.3, enabling accurate, vendor-agnostic assessment of topology discovery efficiency.

Experimental analysis across multiple scalability scenarios demonstrated that OpenDaylight (ODL) consistently outperforms the Open Network Operating System **(ONOS)**, achieving 40–70% faster detection times. Furthermore, ODL exhibited more stable performance growth under increasing topology sizes, underscoring its architectural efficiency in managing topology discovery. These findings establish Graph Detection Time as a critical performance metric and provide actionable insights for network designers deploying SDN in time-sensitive environments.

The developed tool not only contributes a practical mechanism for controller benchmarking but also lays the foundation for broader research. Future work will extend the evaluation to additional controllers such as RYU and Floodlight, investigate the influence of OpenFlow versions beyond 1.3, and validate results in physical testbeds under diverse traffic conditions. Further directions include the development of real-time monitoring dashboards and

resilience testing against Packet-In flooding attacks. These extensions aim to deepen the understanding of SDN controller behavior and enhance the reliability of large-scale production deployments.

REFERENCES

[1]. F. Xu, J. He, and X. Wu, "First study on supply and demand network with multi-functional and opening characteristics for enterprise (SDN)," *Proc. IEEE Int. Conf. on Systems, Man and Cybernetics*, 2004, pp. 1–6.

[2]. M. Sati, M. Emshiheet, A. Majouk, and S. O. Sati, "Control plane comparison of OpenDaylight and Open Network Operating System controllers," *Asia Pacific Advanced Network*, 2024, pp. 1–8.

[3]. R. Narisetty, et al., "OpenFlow configuration protocol: Implementation for the management plane," *Proc. GENI Research and Educational Experiment Workshop*, 2013, pp. 1–4.

[4]. G. Yao, J. Bi, and P. Xiao, "Source address validation solution with OpenFlow/NOX architecture," *Proc. IEEE Int. Conf. on Network Protocols (ICNP)*, 2011, pp. 25–30.

[5]. R. Jmal and L. C. Fourati, "Implementing shortest path routing mechanism using OpenFlow POX controller," *Proc. Int. Symp. on Networks, Computers and Communications (ISNCC)*, 2014, pp. 1–6.

[6]. S. Bhardwaj and S. N. Panda, "Performance evaluation using RYU SDN controller in software defined networking environment," *Wireless Personal Communications*, vol. 122, 2022, pp. 187–200.

[7]. P. Berde, et al., "ONOS: Towards an open, distributed SDN OS," *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014, pp. 1–6.

[8]. J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a model-driven SDN controller architecture," *Proc. IEEE Int. Symp. on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2014, pp. 1–6.

[9]. M. Darianian, C. Williamson, and I. Haque, "Experimental evaluation of two OpenFlow controllers," *Proc. IEEE Int. Conf. on Network Protocols (ICNP)*, 2017, pp. 1–10.

[10]. X. Chen, D. Guo, W. Ma, and L. He, "FloodSight: A visual-aided Floodlight controller extension for SDN networks," *Proc. Int. Conf. on Smart Graphics*, 2015, pp. 100–110.

[11]. I. Z. Bholebawa and U. D. Dalal, "Performance analysis of SDN/OpenFlow controllers: POX versus Floodlight," *Wireless Personal Communications*, vol. 98, 2018, pp. 1879–1895.

[12]. J. Alzarog, et al., "SDN controllers comparison based on network topology," *Proc. Mediterranean Telecommunications Technologies Workshop (MTTW)*, 2022, pp. 1–6.

[13]. Pooja and M. Sood, "SDN and Mininet: Some basic concepts," *Int. J. Advanced Networking and Applications*, vol. 7, no. 2, 2015, pp. 2690–2693.

[14]. B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2010, pp. 1–6.

[15]. D. A. Khoa, et al., "Traffic-adaptive scheme for SDN control plane with containerized architecture," *Proc. Int. Conf. on Ubiquitous Information Management and Communication (IMCOM)*, 2023, pp. 1–7.

[16]. S. Ahmad and A. H. Mir, "Protection of centralized SDN control plane from high-rate Packet-In messages," *Int. J. Information Security*, vol. 22, 2023, pp. 1–16.

[17]. J. B. da Silva, et al., "Benchmarking of mainstream SDN controllers over open off-the-shelf software switches," *Internet Technology Letters*, vol. 3, no. 2, 2020, pp. 1–5.

[18]. M. Boucadair and C. Jacquenet, *Software Defined Networking: A Perspective from within a Service Provider Environment*, RFC 7149, IETF, Mar. 2014.

[19]. B. Vengainathan, et al., *Benchmarking Methodology for Software-Defined Networking (SDN) Controller Performance*, RFC 8456, IETF, Oct. 2018.

[20]. Y. Wang, J. Bi, and K. Zhang, "A tool for tracing network data plane via SDN/OpenFlow," *Science China Information Sciences*, vol. 60, 2017, pp. 1–3.

[21]. J. Alzarog, et al., "POX controller evaluation based on tree topology for data centers," *Proc. Int. Conf. on Data Analytics for Business and Industry (ICDABI)*, 2022, pp. 1–6.

AUTHORS BIOGRAPHY

Tareg A. Abulifa received his BSc and PgD in Computer Engineering from the College of Industrial Technology, Misurata, Libya, and his MSc in Nanotechnology from the University of Glasgow, UK, in 2011. He is currently an academic staff member at the College of Industrial Technology, Misurata. His research interests include computer engineering, nanotechnology, wireless networks, SDN controllers, and IoT systems.

Salem Sati has a PhD in computer science from HHU University in Dusseldorf, Germany (2017), and an MSc & BSc degree in computer engineering from Higher Industrial Institute in Misurata, Libya (2008 & 1997 , respectivally).. Dr. Sati has contributed to several national conferences in Libya, Also International IEEE conferences in North America. Asia and Europe in the areas of computer networks. He has many publications in IEEE conferences. He is currently an assistant professor at the faculty of Information Technology in Misurata University -Libya.