PAPER

Opimon: A Transparent, Low-Overhead Monitoring System for OpenFlow Networks

Wassapon WATANAKEESUNTORN[†a)], **Keichi TAKAHASHI**[†], **Chawanat NAKASAN**[††], *Nonmembers*, **Kohei ICHIKAWA**[†], *Member, and* **Hajimu IIDA**[†], *Nonmember*

SUMMARY OpenFlow is a widely adopted implementation of the Software-Defined Networking (SDN) architecture. Since conventional network monitoring systems are unable to cope with OpenFlow networks, researchers have developed various monitoring systems tailored for OpenFlow networks. However, these existing systems either rely on a specific controller framework or an API, both of which are not part of the OpenFlow specification, and thus limit their applicability. This article proposes a transparent and low-overhead monitoring system for OpenFlow networks, referred to as Opimon. Opimon monitors the network topology, switch statistics, and flow tables in an OpenFlow network and visualizes the result through a web interface in real-time. Opimon monitors a network by interposing a proxy between the controller and switches and intercepting every OpenFlow message exchanged. This design allows Opimon to be compatible with any OpenFlow switch or controller. We tested the functionalities of Opimon on a virtual network built using Mininet and a large-scale international OpenFlow testbed (PRAGMA-ENT). Furthermore, we measured the performance overhead incurred by Opimon and demonstrated that the overhead in terms of latency and throughput was less than 3% and 5%, respectively.

key words: *Software-Defined Networking (SDN), OpenFlow, network monitoring, visualization, networking*

1. Introduction

In the current networking architecture, network devices in a network are individually and manually configured by the administrator. This design makes it challenging to manage large and complex networks. Software-Defined Networking (SDN) [1] is an alternative networking architecture that centralizes the control of network devices to a centralized software controller and introduces programmability to the network infrastructure. In current networks, the packet forwarding function (*data plane*) and the routing decision function (*control plane*) are inseparably implemented in the same network device. In SDN, these two are disaggregated. The packet forwarding is handled by SDN switches, whereas the routing decision is handled by a centralized software controller. Each SDN switch maintains a flow table, which is a collection of flow entries. A flow entry contains a set of (1) matching conditions, which specify the packets that the flow matches, and (2) actions, which specify how matched packets are processed. Flow entries are generated by the controller and installed to switches.

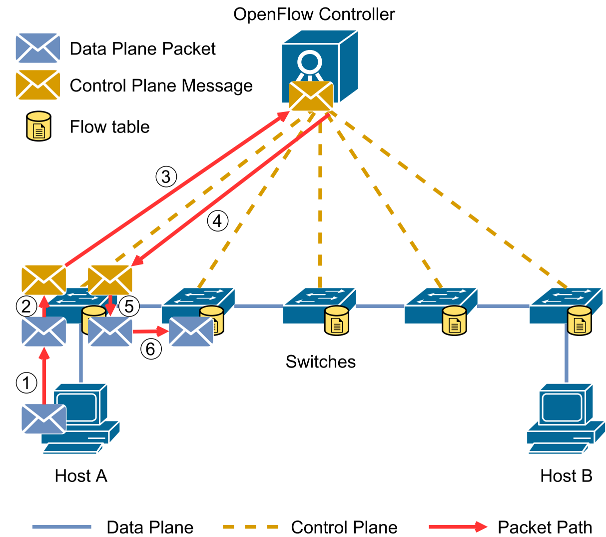


Fig. 1 An OpenFlow network.

The OpenFlow protocol [2] is widely used to communicate between SDN switches and the centralized SDN controller. OpenFlow defines multiple message types for different purposes, such as installing flow entries and collecting switch information and statistics, for example. Some examples of OpenFlow messages are shown in Table 1. Hardware vendors such as Mellanox, Pica8 and NoviFlow produce hardware OpenFlow switches. There are also several software OpenFlow switches including Open vSwitch [3] and Lagopus [4]. Furthermore, software frameworks that facilitate the development of OpenFlow controllers, such as Ryu [5], Faucet [6], Open Network Operating System (ONOS) [7], [8], and OpenDaylight [9], are available.

Figure 1 illustrates how an OpenFlow network delivers a packet. Every time a switch receives a packet from a host (step ① in Fig. 1), the switch searches its flow table for a flow entry that matches the incoming packet (step ②). If a matching flow entry is found, the switch performs the action indicated in the flow entry (step ⑥). If no matching flow entry is found, the switch sends a PacketIn message to the controller (step ③). The controller then examines the PacketIn message and determines where the packet that generated the PacketIn message should be forwarded next. Based on this decision, the controller installs a new flow entry

Manuscript received May 28, 2021.
Manuscript revised September 2, 2021.
Manuscript publicized October 21, 2021.
[†]The authors are with Nara Institute of Science and Technology, Ikoma-shi, 630-0101 Japan.
[††]The author is with Kanazawa University, Kanazawa-shi, 920-1192 Japan.
a) E-mail: wassapon.watanakeesuntorn.wq0@is.naist.jp
DOI: 10.1587/transcom.2021EBP3083

Copyright © 2022 The Institute of Electronics, Information and Communication Engineers

Table 1 Example of OpenFlow messages types.

Message Type	Direction	Purpose
FlowMod	Controller→Switch	Modifies the flow table of a switch.
FeaturesRequest	Controller→Switch	Requests the supported features of a switch.
FeaturesReply	Switch→Controller	Responds to a controller's FeaturesRequest message.
FlowStatsRequest	Controller→Switch	Requests statistics about individual flows on a switch.
FlowStatsReply	Switch→Controller	Responds to a controller's FlowStatsReply message.
PortStatsRequest	Controller→Switch	Requests statistics about individual ports on a switch.
PortStatsReply	Switch→Controller	Responds to a controller's PortStatsReply message.
PacketIn	Switch→Controller	Sends an unmatched packet to the controller.
PacketOut	Controller→Switch	Injects a packet to the data plane of a switch.

to the switch by sending a FlowMod message (step ④). This procedure is repeated until the packet reaches its destination.

Investigating and understanding the behavior of an OpenFlow network is challenging [10], [11]. This is because, although the control logic is logically centralized in the OpenFlow controller, the state of the network (e.g. flow tables) is distributed across the network. Since conventional network monitoring systems are not designed to cope with OpenFlow networks, researchers have developed various monitoring systems tailored for OpenFlow networks [12]–[14]. However, existing systems either rely on a specific controller framework or require modifications to the controller. This is often unacceptable when monitoring production networks.

This article proposes a monitoring system for OpenFlow networks, which we refer to as *Opimon* (OpenFlow Interactive Monitoring)†. Opimon is completely transparent to the network and works with any OpenFlow switch or controller without requiring any modification. Furthermore, Opimon imposes little overhead to the network performance and can be used in production networks. Opimon collects the topology, flow tables, and switch statistics from the target network, and interactively visualizes the state of the network through a web interface in real-time. Opimon is based upon our previous work [15], but its monitoring module is redesigned to minimize the incurred overhead.

Figure 2 compares the design of a conventional OpenFlow monitoring system and Opimon. In a conventional design, the monitoring system was integrated into the OpenFlow controller as a sub component. Thus, the monitoring system was dependent on the OpenFlow controller or the framework it uses. Opimon, on the other hand, acts as a transparent proxy between the controller and switches, and works with any controller. However, this design causes an unavoidable overhead when forwarding and collecting OpenFlow messages. We minimize the overhead by employing a multi-process architecture that scales with the number of switches. Furthermore, we decouple the message forwarding and collection into different processes so that messages are forwarded with minimum delay.

The rest of this article is structured as follows. Section 2 discusses previous works about the monitoring of both conventional networks and SDN. Section 3 explains the design and implementation of Opimon. Section 4 evaluates the performance of Opimon. Finally, Sect. 5 concludes this article and discusses future work.

† https://github.com/wassapon-w/opimon

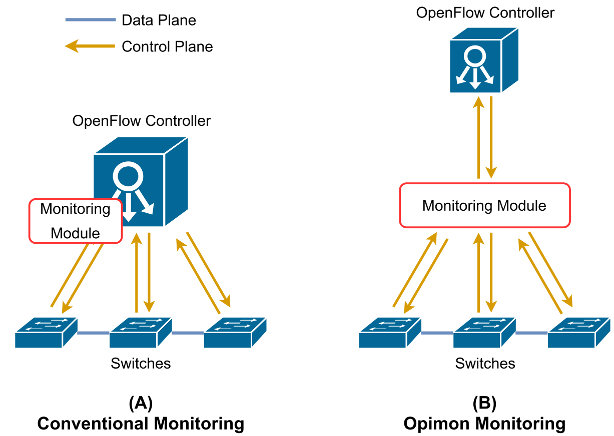


Fig. 2 Comparison between conventional and proposed monitoring system.

2. Related Work

Various monitoring protocols and tools are available in traditional network architectures. Simple Network Management Protocol (SNMP) is one of the most widely used protocols for monitoring networks [16]. SNMP is used to collect information from network devices as well as to configure network devices. sFlow [17] is another popular technology for monitoring the traffic flows in a network. sFlow agents reside on network devices and sample traffic flows from the network, and the sampled traffic is aggregated and analyzed by a sFlow collector. Both SNMP and sFlow are, however, not designed for OpenFlow networks and are unable to obtain OpenFlow-specific information such as the content of flow tables.

Therefore, researchers have designed and implemented monitoring systems tailored for OpenFlow networks. OpenNetMon is a extension module for the POX [18] OpenFlow controller that provides monitoring capabilities [12]. OpenNetMon polls statistics from switches and calculates the throughput and packet loss of each flow. The polling interval is adaptively controlled to reduce the switch CPU load while ensuring measurement accuracy.

OOFMonitor is a monitoring system for OpenFlow net-

works that collects the delay, jitter, packet loss rate, and link utilization [13]. Since OOFMonitor relies on the API exposed by the Ryu OpenFlow controller, it is incompatible with other controllers. In addition, OOFMonitor does not provide any feature to visualize the collected network information.

Isolani et al. proposed a modular system for interactive monitoring, visualization and configuration of OpenFlow networks [14]. Their system uses the RESTful API provided by the Floodlight OpenFlow controller to collect the topology of the network and the traffic counter of every flow entry present on switches.

Warraich et al. developed a system to monitor the traffic statistics at Internet eXchange Points (IXPs), called SDX-Manager [19]. It integrates a traditional IXP-Manager with an SDN controller. Grafana is used to visualize the traffic statistics. However, SDX-Manager is build on top of the Faucet OpenFlow controller framework and lacks support for other controllers.

These existing monitoring systems share a common limitation: they depend on a specific controller or API, which are not part of the OpenFlow specification and not standardized. This limitation clearly hinders practicality because network designers or administrators are forced to choose a specific OpenFlow controller that is compatible with the monitoring system. In contrast, Opimon does not rely on a specific controller or API and can be integrated in any OpenFlow networks.

Network hypervisors such as FlowVisor [20] and AutoVFlow [21] enable virtualization of OpenFlow networks by slicing a physical network into multiple isolated virtual networks. Both of them employ a proxy-based design, where a transparent proxy is placed between the OpenFlow controllers and switches. The proxy examines and modifies the exchanged OpenFlow messages to isolate the network slices with one another. This design allows the hypervisors to be compatible with any OpenFlow controllers or switches. However, monitoring capabilities are not provided.

3. Opimon

In this section, we describe the design and implementation of Opimon. We first describe the high-level architecture of Opimon and then elaborate on each component.

3.1 High-Level Architecture

Figure 3 illustrates the high-level architecture of Opimon. Opimon is mainly composed of two modules: (1) the *monitoring module* and (2) the *visualization module*. The monitoring module behaves as a transparent proxy and intercepts every OpenFlow message exchanged between the controller and the switches. The intercepted messages are stored into a database. Our current implementation uses MongoDB as a database. The visualization module queries the collected messages from the database and shows various network information via a web interface in real-time.

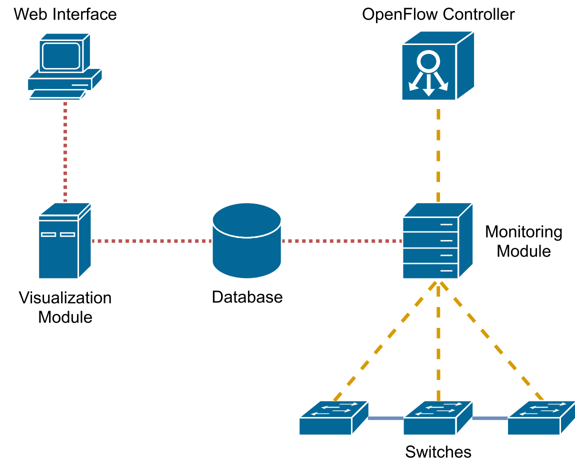


Fig. 3 High-level architecture of Opimon.

3.2 Monitoring Module

3.2.1 Overall Design

The monitoring module is responsible for collecting the OpenFlow messages exchanged in the control plane of an OpenFlow network. Since we found out that messages parsing is the primary bottleneck in collecting OpenFlow messages, we decouple message forwarding and parsing into different processes so that OpenFlow messages can be forwarded with minimal delay. The monitoring module is implemented in Python. This module runs the following three types of processes: connection listener process, message watcher process, and message parser process.

- *Connection listener process*: This process is responsible for handling new connections from switches and coordinating other processes. The connection listener waits for incoming connections from switches and forks a new message watcher process every time a switch is connected. The connection listener also creates a set of message parser processes.
- *Message watcher process*: This process is responsible for forwarding and collecting messages exchanged between the switches and the controller. A message watcher process is created for each switch. Every time a message watcher receives a new message from a switch or a controller, it pushes a copy of the raw message into the message queue and then forwards the message to the other side.
- *Message parser process*: This process is responsible for parsing each message in the message queue and storing the parsed message into MongoDB. This process uses the Ryu OpenFlow framework to parse the raw message. An example of a FlowMod message stored in MongoDB is shown in Listing 1.

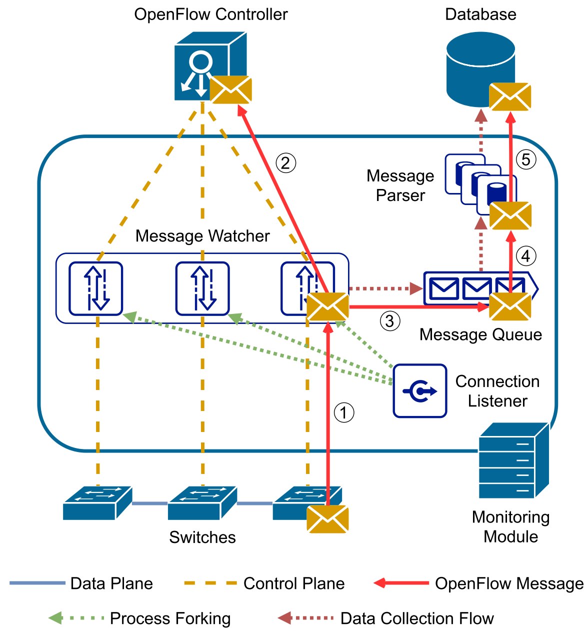


Fig. 4 Monitoring module.

3.2.2 Interaction of Processes

Figure 4 shows the interaction of processes inside the monitoring module. When a switch connects to the monitoring module, the connection listener process forks a new message watcher process. The newly forked message watcher process accepts the connection from the switch and opens another connection to the OpenFlow controller. Every time a message watcher receives a message from a switch (step ① in Fig. 4), the message watcher clones the received message and forwards a copy to the controller (step ②). Another copy of the message is pushed into the message queue (step ③). The message parser asynchronously processes pop messages from the message queue (step ④) and store the parsed messages into MongoDB along with the current timestamp (step ⑤). Messages sent from the controller to switches are handled in the same manner.

The previous version of Opimon [15] employed a single-process and multi-threaded design using Python's threading† module, where the monitoring module launched multiple threads each responsible for receiving, parsing, and forwarding of messages. However, this design suffered from low forwarding performance caused by the Global Interpreter Lock (GIL)†† of Python. GIL is a mutex that ensures only a single Python interpreter thread can execute at a time. Although GIL simplifies the handling of thread-safety, CPU-intensive multi-threaded programs cannot benefit from multi-core CPUs. Using profilers, we found out that message parsing in Opimon is CPU-intensive and blocks the receiving and forwarding of messages. This induced prohibitive latency and packet drops at high traffic load.

In this version, we redesign the monitoring module based on the Python's multiprocessing††† module and separate the collection and parsing of messages into different processes. Since multi-processing is not limited by GIL, the new design allows the monitoring module to utilize multiple CPU cores. In addition, the message watcher processes and the message parser processes are loosely coupled through an asynchronous inter-process queue. This design allows the monitoring module to adapt to sudden changes in the message traffic and to scale the message watchers and parsers independently.


```
"_id" : ObjectId("5f8408271650602248ff3b5d"),
"switch" : "0x2",
"message" : {
    "header" : {
        "version" : 1,
        "type" : 14,
        "length" : 80,
        "xid" : 679114503
    },
    "match" : {
        "wildcards" : 4194294,
        "in_port" : 1,
        "dl_src" : "00:00:00:00:00:00",
        "dl_dst" : "80:00:00:00:00:02",
        "dl_vlan" : 0,
        "dl_vlan_pcp" : 0,
        "dl_type" : 0,
        "nw_tos" : 0,
        "nw_proto" : 0,
        "nw_src" : "0.0.0.0",
        "nw_dst" : "0.0.0.0",
        "tp_src" : 0,
        "tp_dst" : 0
    },
    "cookie" : 0,
    "command" : 0,
    "idle_timeout" : 0,
    "hard_timeout" : 0,
    "priority" : 32768,
    "buffer_id" : NumberLong("4294967295"),
    "out_port" : 65535,
    "flags" : 1,
    "actions" : [ {
        "type" : 0,
        "len" : 8,
        "port" : 2,
        "max_len" : 65509
    } ]
},
"timestamp" : ISODate("2020-10-12T07:39:19.017Z")
```


Listing 1 Example of a FlowMod message stored in MongoDB

3.2.3 Collection of Network Information

In addition to passively intercepting the messages exchanged in the control plane, Opimon actively queries the switches to collect more information. This design, however, causes a side effect because the OpenFlow controller will receive replies to queries that it has not issued. This potentially causes unexpected behavior of the controller and violates the goal of being transparent. Thus, Opimon marks injected messages with a special transaction identifier (xid) to dis-

† https://docs.python.org/3/library/threading.html

†† https://docs.python.org/3/glossary.html#term-global-interpreter-lock

††† https://docs.python.org/3/library/multiprocessing.html

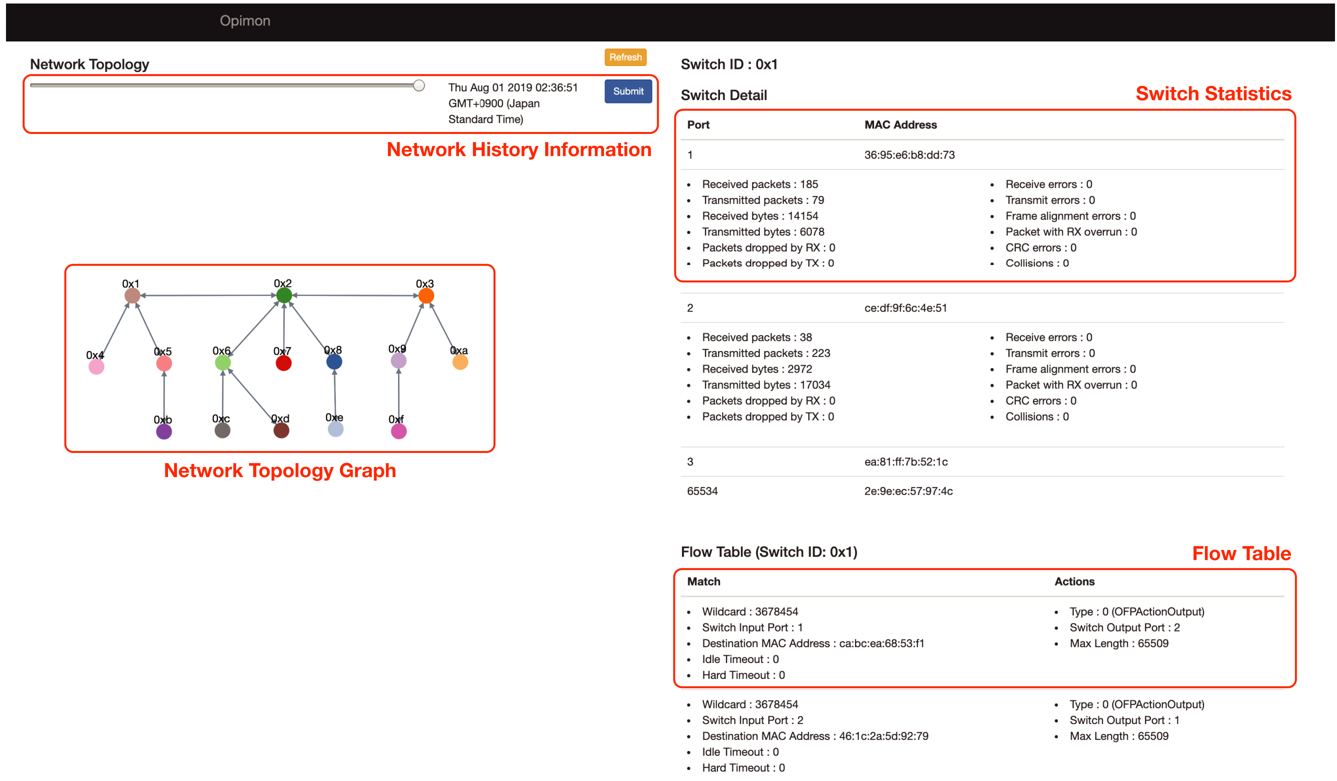


Fig. 5 Visualization of the virtual network using Opimon.

tinguish them from OpenFlow messages generated by the controller. Replies from switches carrying the same special xid are filtered out and not forwarded.

The monitoring module collects the three types of network information in the following manner:

- Network Topology: The network topology is detected using the Link Layer Discovery Protocol (LLDP). The monitoring module injects LLDP packets into a switch using a PacketOut message. When an adjacent switch receives an LLDP packet, it encapsulates the packet in a PacketIn message and sends to the controller. The monitoring module intercepts and parses this message and records the adjacency between switches.
- Switch Information: The switch ID, number of ports, and port MAC addresses are collected by querying the individual switches using FeaturesRequest messages. Port statistics are obtained using PortStatsRequest messages.
- Flow Table: The flow table of each switch is monitored by intercepting FlowMod messages sent out from the controller, which are used to add, modify or delete flow entries on a switch. The statistics of each flow entry is collected by periodically querying switches using FlowStatsRequest messages.

Opimon can detect topology changes in the network caused by incidents such as switch and link failures. When a switch fails and disconnects from Opimon, Opimon stops monitoring the switch and removes it from the web interface. When a link fails, the failed link is detected by LLDP and removed from the web interface.

3.3 Visualization Module

3.3.1 Overall Design

The visualization module is responsible for showing the collected network information to the user in real-time. The visualization module is a web application consisting of a front-end and a back-end.

The front-end periodically polls the back-end to retrieve the latest network information and renders the result as a web page. D3.js is used to render the network topology and jQuery is used to show a table of port statistics and a flow table of the selected switch. The back-end exposes a RESTful API that queries MongoDB and returns the latest network information in JSON format. The back-end is built upon the Express web application framework and Node.js JavaScript runtime.

3.3.2 Web Interface

Figure 5 shows the web interface of Opimon. The web interface has three sections (network topology, switch information, and flow table) divided into two columns.

- Network Topology: This section shows the network topology. A node represents a switch in the network

and an arrow edge represents as a link with a direction of the data flow. Each node is labeled with the ID of the corresponding switch. The labels can be customized in a configuration file to make them easier to identify. Each node in the graph is clickable to show the switch information and flow table of that switch. On top of the network topology, a slider is available to select the time in the past to investigate the previous status of the network that Opimon collected from the selected time.

- Switch Information: This section shows the details of the selected switch in the network topology view as a table. The table shows the MAC address and statistics of each switch port. This information is collected from FeaturesReply message. Each row in the table shows the port statistic that the monitoring module collects from PortStatsReply of StatReply message.
- Flow Table: This section shows the active flows in the selected switch. A table shows the match condition and action of each flow. Hard timeout and idle timeout are shown in the table. The information of the flow table is collected from FlowMod and FlowStatsReply of StatReply messages.

4. Evaluation

We evaluated Opimon from two aspects. First, we deployed Opimon to a virtual network and tested if Opimon can correctly detect the network topology and the flow table of each switch. We conducted the same test on a large-scale international OpenFlow testbed, PRAGMA-ENT. Second, we measured the performance of a controller with and without Opimon and quantified the overhead imposed by Opimon.

4.1 Correctness of Monitoring Results

A virtual network was used to verify if Opimon is able to correctly detect the topology of the network and the flow entries installed on each switch. We used Mininet [22], a network emulator that creates virtual networks comprising many hosts and switches on a single computer, to create a virtual network. Listing 2 shows the Mininet script to create the virtual network. The virtual network comprises 15 switches forming a tree topology. We used Ryu's builtin L2 learning switch (`ryu.ryu.app.simple_switch`) as the OpenFlow controller.

Figure 5 is a screenshot of Opimon's web interface when monitoring the virtual network. We verified that the network topology and the flow entries in each switch are correct.

Opimon was also deployed to a large-scale international OpenFlow network testbed referred to as the PRAGMA Experimental Networking Testbed (PRAGMA-ENT) [23]. This testbed is maintained and used by researchers participating in the Pacific Rim Application and Grid Middleware Assembly (PRAGMA). The OpenFlow switches in this network, including both hardware and software switches, are deployed at multiple PRAGMA partner institutions in

```
from mininet.topo import Topo

class MyTopo(Topo):
  def __init__(self):
    # Initialize topology
    Topo.__init__(self)

    # Create Switch1 to Switch15
    ...

    # Add links between core switches
    self.addLink(Switch1, Switch2)
    self.addLink(Switch2, Switch3)

    # Add links between core and edge switches
    self.addLink(Switch1, Switch4)
    self.addLink(Switch1, Switch5)
    self.addLink(Switch2, Switch6)
    self.addLink(Switch2, Switch7)
    self.addLink(Switch2, Switch8)
    self.addLink(Switch3, Switch9)
    self.addLink(Switch3, Switch10)

    self.addLink(Switch5, Switch11)
    self.addLink(Switch6, Switch12)
    self.addLink(Switch6, Switch13)
    self.addLink(Switch8, Switch14)
    self.addLink(Switch9, Switch15)

topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Listing 2 Mininet script to create the virtual network

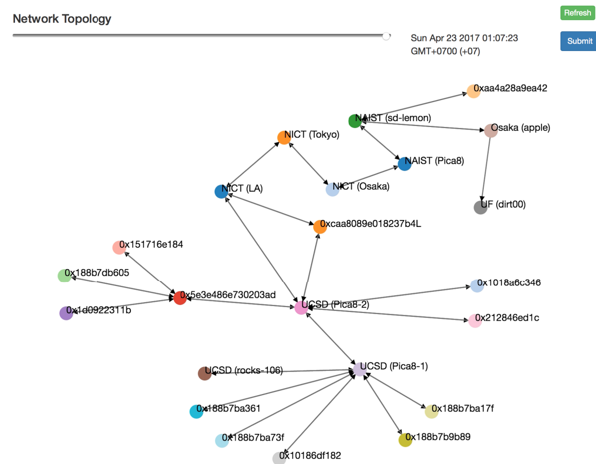


Fig. 6 Visualization of the PRAGMA-ENT network using Opimon.

Japan, the United States, and Taiwan. The switches are connected via VLANs and Generic Routing Encapsulation (GRE) tunnels. PRAGMA-ENT uses a controller implementation based on a routing switch of Trema OpenFlow framework to emulates a layer 2 switch [24]. Figure 6 shows the network topology of PRAGMA-ENT. It shows the switches deployed at the Nara Institute of Science and Technology (NAIST), Osaka University, National Institute of Information and Communications Technology (NICT), University of California San Diego (UCSD), and University of Florida (UF). We confirmed that Opimon was able to correctly monitor the PRAGMA-ENT network in real-time.

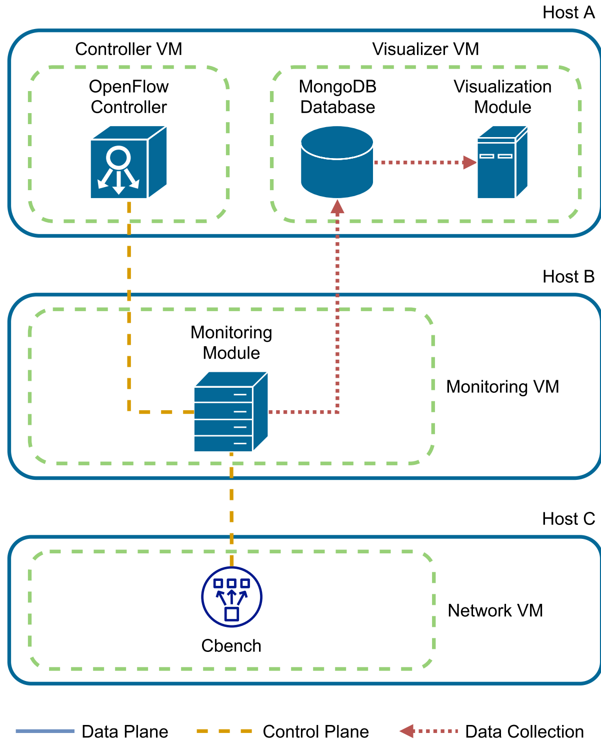


Fig. 7 Experimental environment.

Table 2 Virtual machines used for evaluation.

VM	vCPU	RAM	Software
Controller VM	4	8 GB	Ryu L2 Learning Switch
Visualizer VM	8	16 GB	Visualization Module & MongoDB
Monitoring VM	16	16 GB	Monitoring Module
Network VM	16	16 GB	Cbench

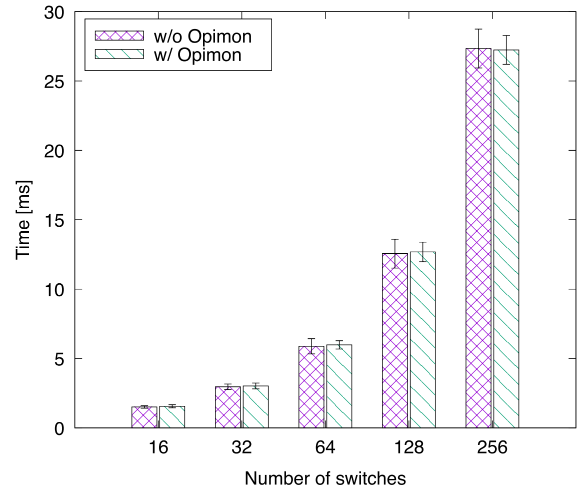


Fig. 8 Controller latency (Ryu L2 learning switch).

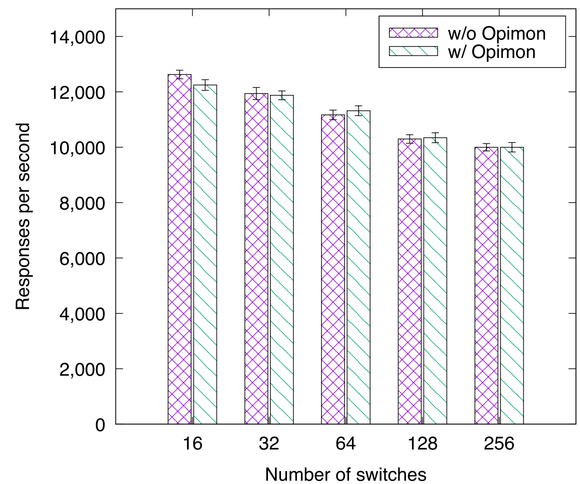


Fig. 9 Controller throughput (Ryu L2 learning switch).

4.2 Overhead Imposed by Opimon

We measured the latency and throughput of Ryu's L2 learning switch controller with and without Opimon to quantify the overhead imposed by Opimon. A benchmark tool for OpenFlow controllers called Cbench [25], [26] was used in this evaluation. Cbench simulates a number of OpenFlow switches by opening multiple connections to the controller and concurrently sending PacketIn messages to simulate the arrival of packets at switches. In the latency mode, Cbench sends a PacketIn message and waits for the controller to reply with a FlowMod message. In the throughput mode, Cbench sends a large number of PacketIn messages and counts the number of FlowMod messages received from the controller.

We set up four VMs on three hosts for this evaluation as shown in Fig. 7. On host A, we deployed a VM that ran the OpenFlow controller and another VM that ran Opimon's visualization module and MongoDB. We deployed a VM for Opimon's monitoring module on host B and a VM for Cbench on host C. All hosts were equipped with two Intel Xeon Silver 4208 CPUs and 96 GB of RAM. Table 2 shows the resource allocation to each VM.

We used Python 3.8.5 and the latest master version of Ryu [5] (git commit `a394673`). The visualization module was executed with Node.js 10.19.0 and Express 4.17.1. MongoDB 3.6 was used as the database. We used Cbench included in the latest master version of Oflops [26] (git commit `762d517`) and built it with the reference OpenFlow implementation [27] (git commit `82ad07d`). All VMs ran Ubuntu Server 18.04.

Using Cbench, we measured the latency and throughput of the controller while varying the number of simulated switches from 16 to 256. We compared the latency and throughput with and without using Opimon in each case. Each measurement was repeated 10 times to quantify the performance variability.

Figure 8 shows a comparison of latency. Here the error bars represent the standard deviation. As expected, the latency of the controller becomes higher when the number of switches increases. The results indicate that Opimon introduces an overhead of approximately 0.5 μs at maximum. This is only 3% of the latency without Opimon, even when

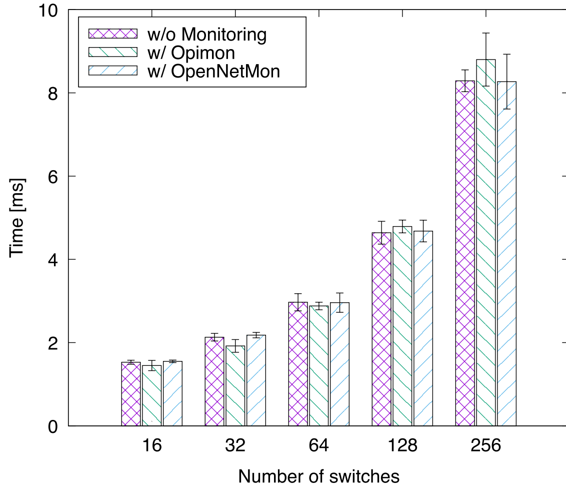


Fig. 10 Controller latency (OpenNetMon routing switch).

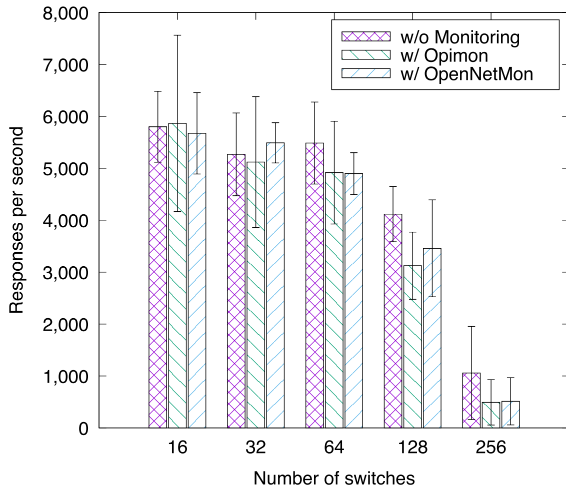


Fig. 11 Controller throughput (OpenNetMon routing switch).

handling 256 switches. Figure 9 shows a comparison of throughput. The results shows that Opimon decreases the throughput of the controller for 5% at most.

4.3 Performance Comparison to OpenNetMon

In this experiment, we compared the overhead of Opimon to an existing OpenFlow monitoring system, OpenNetMon [12]. Here we used the same environment as the previous experiment (Fig. 7), and deployed OpenNetMon on the controller VM. Since OpenNetMon only works with its builtin routing switch controller based on POX, we measured the overhead caused by Opimon and OpenNetMon using this routing switch controller.

Figures 10 and 11 show the latency and throughput measured using Cbench. These plots indicate that the performance difference between OpenNetMon and Opimon is marginal. Furthermore, the fact that Opimon worked with OpenNetMon's routing switch controller based on POX demonstrates that Opimon is transparent to the OpenFlow controller and controller framework.

5. Conclusion & Future Work

We proposed Opimon, a monitoring system for OpenFlow networks. Opimon collects the topology, flow tables, and switch statistics from the target network, and interactively visualizes the state of the network through a web interface in real-time. Opimon is completely transparent to the network and works with any OpenFlow switch or controller without any modification required. Furthermore, Opimon imposes little overhead to the network performance and can be used in production networks. Using Cbench, we simulated up to 256 virtual switches and measured the latency and throughput of the controller with and without using Opimon. The results indicated that the overhead to latency introduced by Opimon is less than 0.5μs (or 3%). In addition, the overhead in terms of throughput was less than 5%.

As future work, we are planning to implement a new module for detecting anomalous traffic in the network, such as DDoS attacks, using machine learning algorithms. In contrast to existing Intrusion Detection Systems (IDS) which require traffic probes to be installed in the data plane, this module will not require additional traffic probes because it will analyze the OpenFlow messages collected using Opimon. We will also extend the visualization module to report the detection results in real-time.

Acknowledgments

This work was partly supported by JSPS KAKENHI (JP18K11326) and ROIS NII Open Collaborative Research 2020 (20S0108).

References

[1] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), pp.1–13, 2013.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, vol.38, no.2, pp.69–74, 2008.

[3] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, and P. Shelar, "The design and implementation of Open vSwitch," 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pp.117–130, 2015.

[4] "Lagopus switch and router," http://www.lagopus.org, accessed May 27. 2021.

[5] "Ryu SDN framework," https://github.com/faucetsdn/ryu, accessed May 27. 2021.

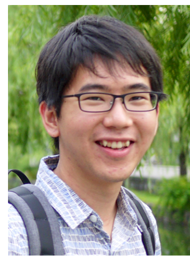
[6] "Faucet: Open source SDN controller for production networks," https://faucet.nz, accessed May 27. 2021.

[7] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an open, distributed SDN OS," Proc. Third Workshop on Hot Topics in Software Defined Networking, pp.1–6, 2014.

[8] "Open network operating system (ONOS)," https://www.opennetworking.org/onos/, accessed May 27. 2021.

[9] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a model-driven SDN controller architecture," Proc. IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014, pp.1–6, 2014.
[10] K. Suzuki, K. Sonoda, N. Tomizawa, Y. Yakuwa, T. Uchida, Y. Higuchi, T. Tonouchi, and H. Shimonishi, "A survey on openflow technologies," IEICE Trans. Commun., vol.E97-B, no.2, pp.375–386, Feb. 2014.
[11] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?," Proc. first workshop on Hot topics in software defined networks, pp.55–60, 2012.
[12] N.L. Van Adrichem, C. Doerr, and F.A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," 2014 IEEE Network Operations and Management Symposium (NOMS), pp.1–8, 2014.
[13] R.B. Santos, T.R. Ribeiro, and C. de A.C. César, "A network monitor and controller using only OpenFlow," 2015 Latin American Network Operations and Management Symposium (LANOMS), pp.9–16, 2015.
[14] P.H. Isolani, J.A. Wickboldt, C.B. Both, J. Rochol, and L.Z. Granville, "Interactive monitoring, visualization, and configuration of OpenFlow-based SDN," 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), pp.207–215, 2015.
[15] W. Wassapon, P. Uthayopas, C. Chantrapornchai, and K. Ichikawa, "Real-time monitoring and visualization software for OpenFlow network," 2017 15th International Conference on ICT and Knowledge Engineering (ICT KE), pp.1–5, 2017.
[16] J. Case, M. Fedor, M.L. Schoffstall, and J. Davin, "RFC1157: Simple network management protocol (SNMP)," 1990.
[17] P. Phaal, S. Panchen, and N. McKee, "RFC3176: InMon Corporation's sFlow: A method for monitoring traffic in switched and routed networks," 2001.
[18] S. Kaur, J. Singh, and N.S. Ghumman, "Network programmability using POX controller," ICCCS International Conference on Communication, Computing & Systems, pp.134–148, 2014.
[19] S.H. Warraich, Z. Aziz, H. Khurshid, R. Hameed, A. Saboor, and M. Awais, "SDN enabled and OpenFlow compatible network performance monitoring system," arXiv preprint arXiv:2005.07765, 2020.
[20] R. Sherwood, G. Gibb, K.K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "FlowVisor: A network virtualization layer," OpenFlow Switch Consortium, Technical Report, vol.1, p.132, 2009.
[21] H. Yamanaka, E. Kawai, S. Ishii, and S. Shimojo, "AutoVFlow: Autonomous virtualization for wide-area openflow networks," 2014 third European workshop on software defined networks, pp.67–72, 2014.
[22] R.L.S. De Oliveira, C.M. Schweitzer, A.A. Shinoda, and L.R. Prete, "Using Mininet for emulation and prototyping software-defined networks," 2014 IEEE Colombian Conference on Communications and Computing (COLCOM), pp.1–6, 2014.
[23] K. Ichikawa, P. U-Chupala, C. Huang, C. Nakasan, T.L. Liu, J.Y. Chang, L.C. Ku, W.F. Tsai, J. Haga, and H. Yamanaka, "PRAGMA-ENT: An international SDN testbed for cyberinfrastructure in the Pacific Rim," Concurrency and Computation: Practice and Experience, vol.29, no.13, p.e4138, 2017.
[24] "Routing switch," https://github.com/trema/apps/tree/master/routing_switch, accessed May 27, 2021.
[25] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A.W. Moore, "OFLOPS: An open framework for OpenFlow switch evaluation," International Conference on Passive and Active Network Measurement, pp.85–95, 2012.
[26] "Cbench: A benchmarking tool for OpenFlow controller," https://github.com/mininet/oflops/tree/master/cbench, accessed May 27, 2021.
[27] "Stanford OpenFlow 1.0 reference switch/controller," https://github.com/mininet/openflow, accessed May 27, 2021.

Wassapon Watanakeesuntorn received the B.Eng. degree in computer engineering from Kasetsart University, Bangkok, Thailand, in 2017 and the M.Eng. degree in information science from Nara Institute of Science and Technology, Nara, Japan, in 2017. In 2019, he was a visiting scholar at University of California San Diego and Salk Institute for Biological Studies, CA, USA. He is currently pursuing the Ph.D. degree in information science at Nara Institute of Science and Technology, Nara, Japan. His research interest includes high-performance computing and networking.



Keichi Takahashi received his M.Sc. and Ph.D. degrees in information science from Osaka University, Osaka, Japan in 2016 and 2019, respectively. In 2018, he was a visiting scholar at the Oak Ridge National Laboratory, Oak Ridge, TN, USA. Since 2019, he has been an an assistant professor at the Nara Institute of Science and Technology, Nara, Japan. His research interests include high performance computing and parallel distributed computing.

Chawanat Nakasan received his B.Eng. in Computer Engineering from Kasetsart University, Thailand, in 2012, followed by Master and Doctor of Engineering degrees from Nara Institute of Science and Technology, Japan, in 2015 and 2018. He currently works as an assistant professor at Kanazawa University in Japan, specializing in cybersecurity, network technology, and computer science education.

Kohei Ichikawa is an Associate Professor in the Division of Information Science at Nara Institute of Science and Technology (NAIST), Japan. He received his B.E., M.S., and Ph.D. degrees from Osaka University in 2003, 2005, and 2008, respectively. He was a postdoctoral fellow at the Research Center of Socionetwork Strategies, Kansai University from 2008 to 2009. He also worked as an Assistant Professor at the Central Office for Information Infrastructure, Osaka University from 2009 to 2012. His current research interests include distributed systems, virtualization technologies and Software Defined Networking.

Hajimu Iida received his B.E., M.E., and Dr. of Eng. degrees from Osaka University in 1988, 1990, and 1993, respectively. From 1991 to 1995, he worked for the Department of Information and Computer Science, Faculty of Engineering Science, Osaka University as a research associate. Since 1995 he has been with the Graduate School of Information Science, Nara Institute of Science and Technology, Japan. His current position is a Professor of the Laboratory of Software Design and Analysis. His research interests include modeling and analysis of software and development process.