Towards ONOS-based SDN Monitoring using In-band Network Telemetry

Nguyen Van Tu, Jonghwan Hyun, and James Won-Ki Hong
Dept. of Computer Science and Engineering, POSTECH, Pohang, Korea
Email: {tunguyen, noraki, jwkhong}@postech.ac.kr

***Abstract*—In the modern era of software-defined networking (SDN), network monitoring is becoming more important in providing information for SDN controllers to control networks. However, current monitoring methods, based primarily on sampling and polling, have performance and granularity limitations. In-band Network Telemetry (INT) is a new method that can perform end-to-end monitoring directly in the data plane, enabling real-time and fine-grained network monitoring. In this paper, we present an INT architecture for the UDP and discuss its design and implementation in Open Network Operating System (ONOS) controller. To the best of our knowledge, this is the first INT monitoring system in ONOS.**

***Keywords*—SDN, Network Monitoring, In-band Network Telemetry, ONOS.**

I. Introduction

Software-defined approaches are currently being introduced in many areas because of their flexibility. In the networking area, software-defined networking (SDN) is being used increasingly. SDN decouples integrated traditional switches into two separated parts - the data plane for forwarding the packets and the control plane for controlling the data plane - usually through an open standard protocol (e.g., OpenFlow). This separation provides SDN with flexibility in controlling network behavior, which a conventional network cannot provide. The Open Network Operating System (ONOS) is one of the widely used mainstream SDN controllers.

Fully exploiting the strength of network control in SDN requires having an efficient network monitoring system. Network information such as real-time traffic, real-time latency, and link changes is essential for an SDN controller to detect anomalies (e.g., traffic spikes) or failures (e.g., terminated links) quickly to enable the controller to take suitable actions. Network monitoring also provides traffic information for an SDN controller to perform traffic accounting or engineering. Thus, an SDN network monitoring system must be real time and fine-grained. In addition, the monitoring process must not use excessive network bandwidth and controller CPU load. However, owing to the limitations of current switches, many of the network monitoring methods in the literature are based on polling and sampling, which limit the granularity and accuracy of monitoring data.

In-band network telemetry (INT) [1] is a new approach to network monitoring. INT performs the monitoring process directly on the data plane, attaching a real-time network state to every packet at the line rate, thereby allowing fine-grained monitoring. However, this approach requires modifications of switches to attach additional data to the original packets. This problem can be solved using Programmable Data Plane with Programming Protocol-Independent Packet Processors (P4) [2]. P4 allows programming of how the packet is processed inside the switches, which is suitable for implementing INT.

This paper presents a network monitoring system for ONOS controller using INT. Our contributions are as follows:

- A proposal of a detailed design INT using UDP as the encapsulation protocol and its implementation in P4. The design and implementation can also be modified easily to support the TCP protocol.
- The first design and implementation of an INT monitoring system for the ONOS controller, bringing packet-level monitoring to ONOS. The application comes with flexible control and monitoring through a graphical user interface (GUI).

The remainder of the paper is organized as follows. Section II covers traditional and current related work regarding network monitoring and the background for our work. Section III presents our modified version of INT for UDP flows and the INT switch. Section IV shows the design of an INT monitoring application in ONOS. Section V presents an evaluation and discusses open issues and possible solutions. Section VI concludes this study.

II. Related Work and Background

A. Network Monitoring methods

1) Traditional: Two traditional methods, NetFlow [3] and SFlow [4], have been widely used for many years. NetFlow performs per-flow monitoring. It captures the information of Internet Protocol (IP) flows when they pass through switches, and then exports the aggregated data. NetFlow thus requires additional memory and workload on a switch CPU for extracting and processing flow data. In addition, to obtain report data, a monitoring center need to perform polling on the switches, and since the shortest period for exporting NetFlow data is 15 s, NetFlow is not well suited for real-time monitoring.

SFlow, or Sample Flow, samples the packet flows passing through the network interface. SFlow takes a sample once every n packets, with the sampling rate n being configurable. The sampled packet can then be sent immediately to a target

Authorized licensed use limited to: Frankfurt University of Applied Sciences. Downloaded on July 14,2022 at 15:56:35 UTC from IEEE Xplore. Restrictions apply.

instance for further analysis. SFlow requires little additional CPU and memory on switches, but has a disadvantage with regard to accuracy. The sampling method might miss network spikes or anomalies or be unable to detect small flows.

2) SDN - OpenFlow based: With the separation of the control and data planes, new methods for monitoring developed in an SDN environment have recently been introduced. Many of these are based on OpenFlow, the de facto most widely used SDN protocol. Yu et al. [5] proposed a push-based monitoring approach, rather than a polling-based method. FlowSense leverages *PacketIn* and *FlowRemoved* messages sent from switches to a controller to report the network state. It has very low overhead at the cost of not being able to perform real-time and accurate monitoring.

OpenNetMon [6] uses a polling technique instead. The polling rate is adaptive, increasing when flow rates change rapidly and decreasing when they stabilize to minimize the number of queries. Adaptive polling provides reasonable accuracy while maintaining low CPU overhead. However, since OpenNetMon polls only the edge switches, it does not provide information regarding intermediate switches and lacks a complete view of the network state.

OpenSample [7] is another monitoring framework for the OpenFlow protocol that leverages SFlows approach of sampling the packets. Thus, the focus of OpenSample is traffic engineering, since it can quickly detect elephant flows and estimate link utilization.

3) Programmable Data Plane based: All of these systems are constrained by fixed function switches. Recently, a programmable data plane has been proposed, changing switches from fixed functions to programmable ones. Some new methods that utilize this ability have been proposed, such as OpenSketch [8], UnivMon [9], and INT. The OpenSketch and UnivMon processes sketch-based streaming algorithms inside switches, allowing fine-grained monitoring with low overhead. OpenSketch is deployed in FPGA switches, while UnivMon is deployed using P4.

INT is a monitoring framework designed to collect and report network states directly from the data plane. The idea is to attach the information to packets that run through network devices, and then extract it at the end switch and send this information to a monitoring center [10]. The process is performed at full line rate without delaying the packets. Control parameters, e.g., which information to collect, are set up as telemetry instruction from a central controller to network devices. Thus, INT can provide end-to-end packet-level network information in real time, in contrast to previous methods, which cannot. In addition, INT provides a complete view of the network, again in contrast to other methods, which require polling or sampling of all switches. Thus, INT enables many applications such as network troubleshooting, advanced congestion control, advanced routing, or network data plane verification [11]. Moreover, since all work is performed on the data path, it can be expected to cost little in switch CPU as well as in central controller CPU for polling information (though it does require a central data processing for further INT data analysis). Compared to OpenSketch and UnivMon, INT is more mature and defined with a specification [11].

B. P4 and ONOS

P4, a high-level language for programming protocol- independent packet processors, is used to program how packets are to be processed in the data path, e.g., in P4-supported switches. In the original switch, the specification already indicates what a switch can do and what it cannot do. Even the OpenFlow switch still has a fixed set of functions mentioned in the OpenFlow specification. This leads to very complex switch design when the OpenFlow specification grows and supports more network protocols. P4 solves this problem by directly programming switch data paths, allowing programmers to decide how the switch processes packets with custom actions and packet formats.

ONOS [12], unlike many central SDN controllers, uses a distributed approach: we have several ONOS controller instances instead of only a central one. Thus, ONOS has high scalability and availability. ONOS also allows creating new services easily through abstraction. From version 1.6 (Goldeneye), ONOS supports the P4 language with a BMv2 [13] switch, a P4 software switch.

III. INT Switch

A. Data format

The format of INT is detailed in the INT specification [11]. In general, a packet with INT information is the original network packet with two more fields: the INT header, which contains all fixed INT parameters and information, and the INT data, which contain per-switch information that is added every time an INT packet passes through a switch. These fields must be attached as an option or payload in an encapsulated protocol. INT does not indicate a detailed format for any specific protocol but provides an example of INT as the VXLAN option. This paper proposes INT as a shim header in UDP packets.

```
|       0       |       1       |       2       |       3       |
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |           INT Port            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ UDP
|            Length             |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver|Rep|c|e|O|r r r r| Ins Cnt |  Max Hop Cnt  | Total Hop Cnt|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+INT
|      Instruction Bitmap       |           Reserved            |header
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Int len            |      Original Dest Port       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        INT data Stack                         | INT
|                                                               | data
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Original payload                        |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 1. INT format for UDP protocol

Authorized licensed use limited to: Frankfurt University of Applied Sciences. Downloaded on July 14,2022 at 15:56:35 UTC from IEEE Xplore. Restrictions apply.

Fig. 1 shows the UDP packet format with the INT header and data as a shim header. It is inserted right after the original UDP header and before the UDP payload. The detailed information of common fields is well defined in the INT specification. Beside, there are four custom fields defined for INT in UDP:

- *INT Port* and *Original Dest Port*: For UDP flows, INT requires a specific port, *INT port*, to differentiate it from other protocols. We make the INT port replace the original UDP destination port. When switches acknowledge the INT port, it knows that the packet comes with INT instructions and performs the next INT process. The original UDP destination port is backed up at the field *Original Dest Port*.
- *O*: This is an additional field for informing the ONOS controller whether the INT packet is sent to the controller for analysis (*O* field is one) or is just a normal packet with INT data that must be forwarded (*O* field is zero).
- *INT len*: the total length of INT header and data.

The INT header has a fixed length of 12 bytes, while the INT data stack carries the monitoring data with variable length. The same approach can be used to support the TCP as well.

B. Switch design

The general architecture of a P4 switch is shown in Fig. 2. Input packets are first deserialized (parsed) to *Parser Representations*, which represent packet structures. Then, packets pass through *Ingress* for modifications and selections of the Egress port. Next, packets are pushed to *Queues/Buffers* before passing through *Egress* for further required modifications. Finally, packets pass through *Deparser* for serialization and exit the switch. For the current version of P4, the *Deparser* uses the packet structure described in *Parser* to serialize packets, and is thus not yet programmable.

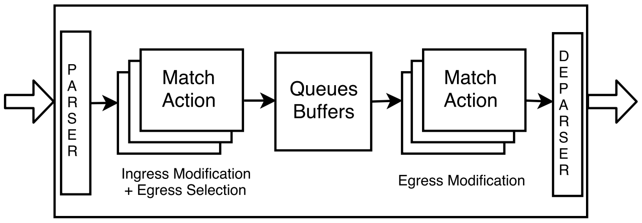

Fig. 2. P4 Switch architecture [14]

P4 is designed using two key ideas: 1) match + action tables for simplification and compatibility and 2) pipeline processing for improved throughput. *Match/Action* tables are put in *Ingress* and *Egress*. P4 packet processing can be described as a flow of packets running through the tables. A table compares the matching fields in flow entries with packet fields or metadata to determine the appropriate action (add/remove/change fields or passing the table without any action). After finishing the modification, packets pass to the next table for the next match/action process.

The processing flow of an INT switch is shown in Fig. 3. We use P4_14, version 1.0.3 [14], a widely used version of P4. Following the general P4 design, the processing flow of an INT switch contains three principal parts: Parser, Ingress, and Egress, of which Ingress performs forwarding and determines the necessary flags and Egress performs most of the work of INT processing.

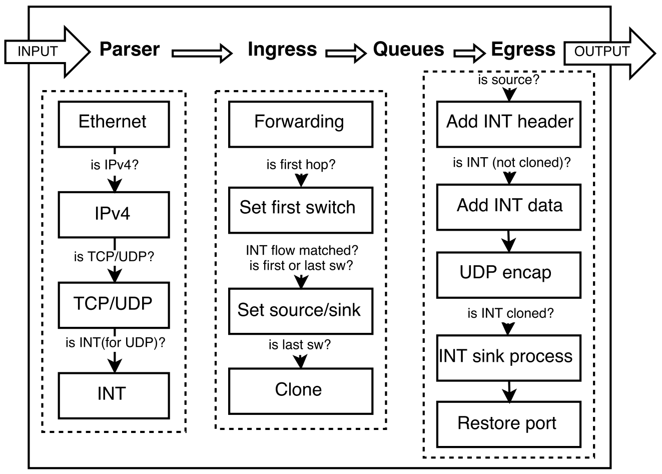

Fig. 3. INT Switch processing flow

1) Parser: The switch design supports INT with the basic TCP/UDP protocol. The parser is thus required to parse only Ethernet, IPv4, TCP, and UDP/INT headers. At each parsing state, if the condition is satisfied, packets will pass to the next stage. Otherwise, the parsing process finishes.

2) Ingress: After parsing, packets are fed to Ingress match/action for port selection. A basic forwarding match + action table is implemented in Ingress, which supports basic ONOS services and applications such as link layer discovery and reactive forwarding. These enable the transmission of UDP packets from sources to destinations through the network.

Ingress is also where some metadata for packets is extracted. First, the switch is checked to determine whether it is the first hop of the packet (applied for every packet, not only INT packets). Next, Ingress processes determine whether the switch is the source (first switch in the path) or the sink (last switch in the path), and then set the equivalent flag. In addition, if the switch is the sink switch, the packet is cloned. After completing Ingress, the packet is sent to Queues/Buffers.

3) Egress: After exiting queues, packets pass through the Egress Match + Action for attaching of INT headers. A series of match/action tables is applied. First, Egress checks the source flag. If the source flag is set, an INT header is inserted into a UDP packet using telemetry instructions sent from the controller. Next, INT information for this switch

Authorized licensed use limited to: Frankfurt University of Applied Sciences. Downloaded on July 14,2022 at 15:56:35 UTC from IEEE Xplore. Restrictions apply.

is attached at the end of the INT data (after the data of the last switch). The type of monitoring data is determined by telemetry instructions. Finally, the outer UDP header is updated. Packets exiting Egress are sent to the Deparser to be serialized, and are then sent out of the switch.

In a case in which a switch is the last switch before reaching the destination host, packet mirroring is enabled. The entire packet with its INT metadata is cloned. Now there are two identical copies of the original packet. For one packet, Egress attaches the last INT data of the sink switch and sends the packet to the controller. The remaining packet has all of the INT data and the header removed (INT sink process) and the original destination port restored, and is then sent to the destination. In this manner, the monitoring system becomes transparent to end hosts.

Remember that all of the blocks in Ingress and Egress are match/action tables. Although P4 supports branching (e.g., If - else) code, to simplify the design of a switch, we choose to push almost all decision logics into table rules and these rules will be installed by the controller right after the initialization.

IV. ONOS INT Design

Fig. 4 shows the architecture of the INT monitoring system for ONOS. ONOS controller communicates with BMv2 switches through *Thrift* protocol, a light-weight, general and flexible protocol which is suitable with the flexibility of P4 language. ONOS INT monitoring system works through three phases: Initial Configuration, Control and Monitoring.

A. Initial Configuration

The BMv2 software switch is just a software simulator for P4 hardware, that is, it requires a P4 configuration file to be fully functional. In this case, the configuration file is the INT switch in json format, *sw.json*, the result after compiling the INT P4 source code. At the beginning, INT Monitoring (or IntMon) controller reads *sw.json* and pushes the configuration data to all BMv2 switches.

B. Control

IntMon controller is in-charge of populating all flow rules to tables in the INT switches. It builds custom flow rules and sends those rules to designated match/action tables in switches. This is performed the first time after the initial phase and every time we change, add, or remove a rule through the GUI. *IntMon controller* must also update the flow rules when there are network events, for example, when hosts are unplugged from, or new hosts are plugged into, the network. The rules for setting up which flows and which fields to monitor are entered through a GUI in *Web interface*. Multiple rules can be stored at a time, and all of the flows that match one of the matching rules are monitored. The flow selections follow the five-tuples rule: protocol (in this implementation, only UDP is allowed), source/destination port, and source/destination IP address. An empty field matches all values for the field. Source and destination IP address fields support both exact matching

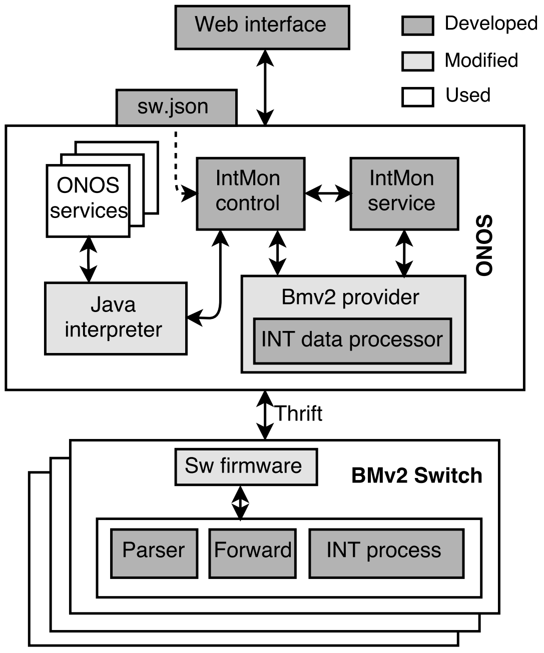

Fig. 4. ONOS INT monitoring architecture

and prefix matching. Because of the prefix matching, it is possible that a flow matches more than one rule. Thus, we also add input for a priority field, with each rule requiring a unique priority value.

Monitoring fields are also selected from the same GUI interface. There are eight network parameters that can be collected: Switch ID, Ingress port, Hop latency, Queue occupancy, Ingress timestamp, Egress port, Queue congestion status, and Egress port utilization. All of the fields have packet-level granularity. We must modify *Switch firmware* in BMv2 to support the collection of all this information.

C. Monitoring

INT packets sent to ONOS are processed at the low-level layer of the ONOS core, *Int data processor* in *BMv2 provider*. Typically, a packet sent to ONOS is passed to all upper applications that have packet requests. Because we expect the number of INT packets to be huge, we decide to process and then discard INT packets as soon as possible to prevent them from being processed by other applications, thereby reducing the CPU and memory cost of the ONOS controller.

INT data are analyzed in *IntMon service* and can then be displayed in *Web interface*. We provide a time series graph to show the hop latency of a switch in real time, updated ten times per second; a tabular view for showing all monitoring parameters, updated once per second; and a topology view to update link bandwidth. The update interval is limited owing to the CPU cost of refreshing the GUI view. For accessing packet-level information, *IntMon service* provides interfaces for other applications to obtain real-time data.

Authorized licensed use limited to: Frankfurt University of Applied Sciences. Downloaded on July 14,2022 at 15:56:35 UTC from IEEE Xplore. Restrictions apply.

Another important component is *Java interpreter*. Since we have a custom P4 switch with custom rules, the original ONOS services cannot understand our rules. *Java interpreter* provides the rule-interpreting interface to enable basic ONOS services and applications (such as link layer discovery and reactive forwarding) to work without modification.

V. Evaluation and Discussion

We evaluate the performance and overhead of our monitoring system by measuring the IntMon processing time, CPU usage and calculating the network bandwidth cost. We use a simple tree topology in Mininet [15] for testing, as shown in Fig. 5. All the switches are BMv2 switch and connected with ONOS controller through Thrift protocol.

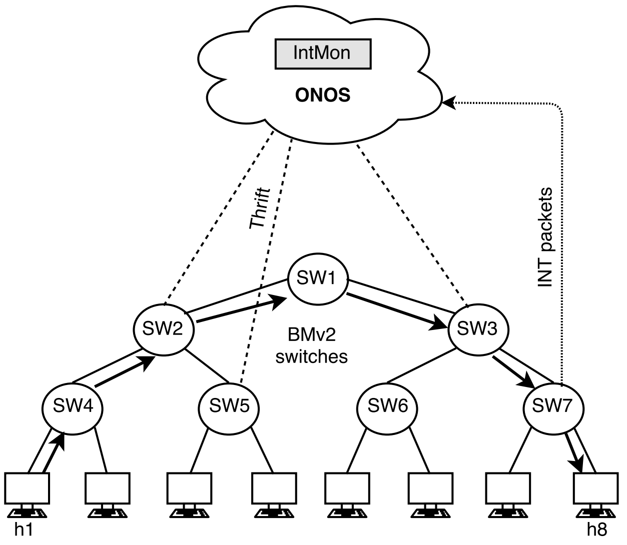

Fig. 5. Evaluation topology

We use iPerf [16] to send UDP flows from host *h1* (client) to host *h8* (server) with total flows from four to 14, 100 kbps per flow, and a data packet size of 80 bytes (the throughput is limited due to the low-performance BMv2 switches). We enable INT monitoring for all these flows. After reaching *sw7*, INT data is sent to ONOS controller, and original packets are forwarded to the destination (*h8*). The ONOS controller runs in a virtual machine (VM) with Ubuntu 14.04 in VirtualBox. The VM has one core CPU and 2 GB RAM.

A. Average processing time and CPU usage

The average CPU usage of the ONOS controller depends on the number of INT packets, as shown in Fig. 6. In addition, Table I shows the average delay time of some common ONOS services, including the IntMon service.

From Table I, we can see that the IntMon process is much faster than other ONOS services. However, From Fig. 6, we notice that the CPU usage increases linearly with the number of INT packets sent to the ONOS core. Although INT allows the ONOS controller to be free from polling, the CPU is still required for analyzing the massive amount of INT data. We can use a faster CPU for handling INT packets (the simulation context uses a VM with a weak one-core CPU), but the monitoring system still faces the scaling problem.

TABLE I
Average processing delay time

Service	Average delay (ms)
Link Layer Discovery (LLDP)	0.13303
Reactive Forwarding (FWD)	0.23605
INT Monitoring (IntMon)	0.00773

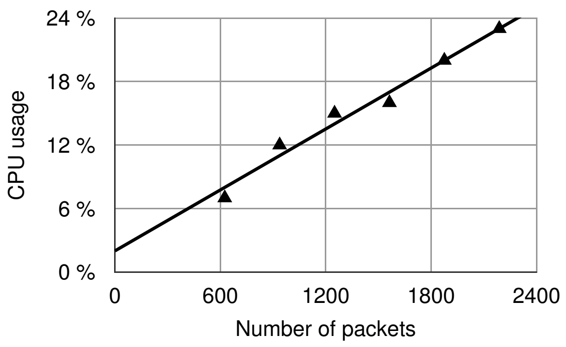

Fig. 6. CPU usage against number of INT packets

B. Bandwidth overhead

Regarding the additional bandwidth for carrying INT data, the formula is:

$$B = P \times (12 + N \times F \times 4) \times 8\,(bps)$$

where B is the additional bandwidth on the link, P is the number of packets (INT enabled) per second, N is the number of switches that the packet has passed through, and F is the total number of monitoring fields. For example, suppose we have a link with two UDP flows sharing equal bandwidth. The link is the TX link of the third switch. All flows are UDP flows with 80-byte data size, meaning a 126-byte packet size (including Ethernet, IPv4, and UDP headers). One of two flows is monitored, and the total number of monitoring fields is three. Then, the application throughput is reduced by 16% (note that the application throughput decreases, but the maximum link bandwidth still remains the same). Even when we do not enable monitoring for all flows, application throughput reduction is still a problem when detection of elephant flows is necessary.

C. Discussion

Evaluation results reveal two obstacles to useful and practical large-scale use of INT, the post-data analysis cost in CPU and memory, and the additional bandwidth cost for INT data. There are methods that can be applied to solve or mitigate these problems. One direction is to offload a portion of the INT analysis to P4 logic instead of sending all information to

Authorized licensed use limited to: Frankfurt University of Applied Sciences. Downloaded on July 14,2022 at 15:56:35 UTC from IEEE Xplore. Restrictions apply.

the central ONOS application. This method reduces CPU cost for processing INT data and is particularly useful for detecting significant network events such as path changes or spikes [17]. Another direction is to move the INT analysis block currently in ONOS to an external INT Collector system.

Regarding the bandwidth overhead, we can reduce the bandwidth cost by deciding to monitor only useful flows. In the event that we must monitor many flows or elephant flows, we can use a sampling approach with a trade-off between granularity level and bandwidth usage, with options to allow enable/disable as well as to change the sampling rate. Sampling might be suitable for some specific monitoring requirements, such as link utilization. In addition, a database for storing monitor data is necessary for advanced queries in a network state.

VI. Conclusion

This paper presents our design and implementation of an INT monitoring framework in the ONOS controller. We proposed an INT architecture for the UDP that can be easily extended for the TCP, and implemented it using P4. We designed and implemented the first INT monitoring system in ONOS that enables packet-level monitoring. The application provides a GUI interface to control and monitor network states. The evaluation results revealed some limitations of our design, and consequently, we proposed some possible solutions for future investigation.

Acknowledgment

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government(MSIT) (No. 2017-0-00195, Development of Core Technologies for Programmable Switch in Multi-Service Networks).

References

[1] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2015.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[3] B. Claise, "Cisco systems netflow services export version 9," *RFC 3954 (Informational), Internet Engineering Task Force*, 2004.

[4] M. Wang, B. Li, and Z. Li, "sFlow: Towards resource-efficient and agile service federation in service overlay networks," *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pp. 628–635, 2004.

[5] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "FlowSense: Monitoring network utilization with zero measurement cost," *International Conference on Passive and Active Network Measurement*, pp. 31–41, 2013.

[6] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks," *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pp. 1–8, 2014.

[7] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "OpenSample: A low-latency, sampling-based measurement platform for commodity SDN," *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pp. 228–237, 2014.

[8] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," *10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, vol. 13, pp. 29–42, 2013.

[9] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pp. 101–114, 2016.

[10] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: Using packets for low latency network programming and visibility," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 3–14, 2015.

[11] C. Kim, P. Bhide, E. Doe, H. Holbrook, A. Ghanwani, D. Daly, M. Hira, and B. Davie, "Inband Network Telemetry," June 2016.

[12] 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," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. ACM, 2014, pp. 1–6.

[13] BMv2, https://github.com/p4lang/behavioral-model.

[14] The P4 Language Consortium, "The p4 language specification 1.0.3," Nov 2016.

[15] Mininet, https://github.com/mininet/mininet.

[16] IPerf, https://github.com/esnet/iperf.

[17] T. Jithin and L. Petr, "Tracking packets paths and latency via INT (In-band Network Telemetry)," *P4 Workshop*, May 2016.

Authorized licensed use limited to: Frankfurt University of Applied Sciences. Downloaded on July 14,2022 at 15:56:35 UTC from IEEE Xplore. Restrictions apply.