Received: 12 February 2019 | Revised: 28 April 2019 | Accepted: 9 May 2019

DOI: 10.1002/nem.2080

RESEARCH ARTICLE

WILEY

Real-time and fine-grained network monitoring using in-band network telemetry

Jonghwan Hyun[1] | **Nguyen Van Tu**[1] | **Jae-Hyoung Yoo**[2] | **James Won-Ki Hong**[1]

[1]Department of Computer Science and Engineering, POSTECH, Pohang, Korea

[2]Graduate School of Information Technology, POSTECH, Pohang, Korea

Correspondence
Jonghwan Hyun, Department of Computer Science and Engineering, POSTECH, 37673, Pohang, Korea.
Email: noraki@postech.ac.kr

Funding information
Institute for Information & communications Technology Promotion, Grant/Award Number: 2018-0-00749

Summary

In the modern era of software-defined networking (SDN), network monitoring is becoming more important for providing information about a network and helping SDN controllers to make decisions about the network. In-band Network Telemetry (INT) is a new network monitoring framework that collects packet-level network information to provide real-time and fine-grained network monitoring. In this paper, we present the design of the overall INT management architecture and its two main components: the INT management system and INTCollector. The INT management system controls heterogeneous INT-capable devices through a common interface. INTCollector is a high-performance collector for INT data, which uses eXpress Data Path and an event detection mechanism. The evaluation result shows that INTCollector processes telemetry reports 27 times faster than other packet-level telemetry collectors. We made the implementation as open source, to make researchers who are interested in INT implement their own ideas on top of our work.

1 | INTRODUCTION

Operations, administration, and maintenance (OAM) are a toolset for fault detection and isolation and performance measurement.[1] Simple Network Management Protocol (SNMP) is one of the most popular protocols for monitoring networks; however, it has performance limitation because of its polling-based nature and high processing overhead. In-band Network Telemetry (INT)[2] is an in-band measurement technique that is designed to collect and report network states directly from the data plane. It provides network visibility through collecting fine-grained network data for analysis and measurement.[3] Network telemetry is based on in-band measurement, in which telemetry information is embedded into data packets as they traverse the network. The embedded telemetry information is associated with the packet that carries the information. It does not require extra packets to collect network status information and, hence, does not change packet traffic mix.

INT traffic sources (source switch in INT) inserts INT headers into packets. INT instruction is embedded in INT header, indicating which type of network information (eg, switch ID and hop latency or link utilization) to collect at each INT-capable switch. When a packet with an INT header is being forwarded in an INT-capable switch, the switch attaches its own network information to the INT header. At a sink switch, an INT telemetry report packet that encapsulates the collected INT information is created and sent to an INT collector. In this way, INT can provide real-time, end-to-end network information with packet-level granularity.

To realize an INT-based network management architecture, we need the following components:

- INT-capable data plane: implements the INT specification.
- INT management system: orchestrates generation and collection of INT data.
- Telemetry collector: stores and processes collected INT data.

In our previous work,[4] we presented a network monitoring system for the Open Network Operating System (ONOS) controller using INT. However, it suffers from two problems. First, it has a low packet processing rate and limited scalability because all INT data collected in data planes are sent to ONOS instances. Second, the system has to populate flow monitoring rules in INT switches and is fitted to the specific INT-capable data plane implementation. Since INT specification[5] does not describe detailed data plane structure and INT packet processing procedure, an implementation of INT-capable data plane may vary.

To overcome these problems, we present the design and implementation of an INT-capable network management architecture, which is composed of an INT-capable data plane, an INT management system and a high-performance collector. The management system controls heterogeneous INT-capable devices and runs on top of an SDN controller. The management system also defines a driver interface for INT-capable devices. Each device driver needs to implement the interface to be controlled by the management system. It also exposes a northbound API for applications to control INT behaviors in the network. We have implemented the management system on ONOS controller.[6]

The INT-capable data plane implements the following logic for processing INT packets: parse INT headers, inject an INT header and INT data, and generate a telemetry report and forward it to a collector. We have designed and implemented the INT-capable data plane using P4.[7]

INTCollector is a high-performance collector for INT that has two separate processing paths: a fast path to process INT report packets and a normal path to process events and store INT data in a database. An event is defined as an important network state change (such as a new flow or a significant change in hop latency), which is extracted from INT telemetry reports by our event detection mechanism. Because INT data are stored only when an event happens, CPU usage and storage costs are reduced while still guaranteeing that important information is captured and stored. Based on our calculations, INTCollector can reduce the amount of data stored by two to three orders of magnitude. The fast path is accelerated with eXpress Data Path (XDP)[8]—an in-kernel fast packet processing framework. In our evaluation, INTCollector can process INT report packets at a rate of 10 Gbps with an average CPU usage of 14.50% when run on a commodity server and hardware NICs, with data plane acceleration technique. It also shows 27 times better performance than other collection methods.[9] As a result, we designed and implemented real-time and fine-grained network monitoring system using INT.

This paper is organized as follows. In Section 2, we provide background for our work, techniques in the literature for network monitoring, fast packet processing, and related work about collectors for INT. We present the detailed design of the INT management architecture and INT management system in Section 3. Section 4 presents implementation details of the proposed architecture. Section 5 shows evaluation results of INTCollector, and Section 6 discusses limitations, design choices, and possible use cases of our work. Section 7 concludes this paper with a summary and future work.

2 | BACKGROUND AND RELATED WORK

In this section, we first present background that is relevant to our work. Second, we survey various techniques for network monitoring and show that INT is superior to the other methods. Finally, we survey related work on collectors for INT.

2.1 | Background

2.1.1 | P4

P4[10] is a high-level language for programming protocol-independent packet processors, and it is used to program how packets are processed in the data path, eg, in P4-supported switches. In traditional switches, the specification indicates what a switch can and cannot do. For example, an OpenFlow switch has a fixed set of functions defined in the OpenFlow specifications. This leads to very complex switch designs as the OpenFlow specification grows and supports more network protocols. P4 solves this problem by directly programming switch data paths, which allows programmers to decide how a switch processes packets using custom actions and packet formats.

Figure 1 shows one common architecture of a P4 switch. A general procedure for a pipeline is as follows:

1. The *Parser* extracts bits from packet headers (eg, Ethernet, IPv4, or TCP).
2. Packets pass through the *Ingress pipeline* for packet header modifications and egress port selection.
3. Packets are pushed to *Queues/Buffers* for queueing, packet scheduling, and packet replication.
4. Packets pass through the *Egress pipeline* for further modifications.
5. Packets pass through the *Deparser* for serialization and then exit the switch.

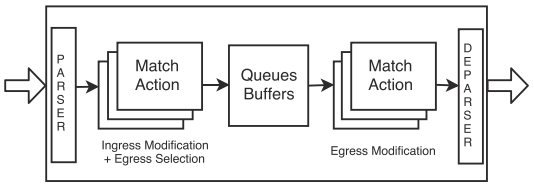


FIGURE 1 P4 switch architecture[7]

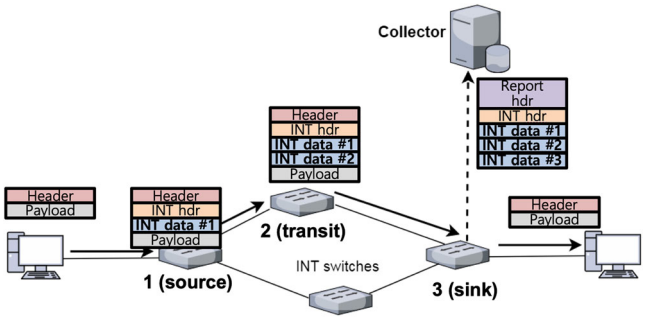


FIGURE 2 INT working process

```
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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Ver  |Rep|C|E|M|     Reserved      | Hop ML  |RemainingHopCnt|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Instruction Bitmap       |            Reserved           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| INT Metadata Stack (Each hop inserts Hop ML * 4B of metadata) |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               . . .                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Last INT metadata                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

FIGURE 3 INT header format[5]

2.1.2 | In-band Network Telemetry

In-band Network Telemetry (INT)[2] is a network monitoring framework designed to collect and report network states directly from the data plane (Figure 2). INT data can be any data provided by switches, such as a timestamp, hop latency, or link utilization. The format of an INT header is depicted in Figure 3.

INT enables many advanced applications, such as network troubleshooting, advanced congestion control, advanced routing, and network data plane verification.[5] INT is implemented with a programmable data plane, which means network functions (switches) can be re-programmed and implementations can vary.[2,4,11] P4 provides a protocol-independent API[12] to control the data plane at runtime, but it is still pipeline-dependent. Therefore, a pipeline-independent API needs to be defined to control heterogeneous INT-capable data planes.

2.1.3 | eXpress data path

eXpress Data Path (XDP)[8] is a kernel framework that allows packet processing inside Linux kernel. Unlike kernel modules that affect the stability of the kernel, XDP is safe and secure. XDP programs process packets immediately after they arrive at a NIC to achieve high throughput. In this way, XDP avoids cost of kernel networking stack processing and also avoids kernel-user space switching.[13]

However, XDP has a limitation. XDP has a restricted programming capability to ensure the safety of the kernel. One such restriction is limited number of instructions. Thus, XDP should be used only for tasks that are simple but require real-time processing. We solved this problem by partitioning INTCollector into a normal path and a fast path. Only fast path is implemented with XDP, as described in Section 3.

Other than XDP, there are other frameworks for fast packet processing, such as Data Plane Development Kit (DPDK).[14] However, XDP has several advantages compared with DPDK:

- XDP does not require a dedicated CPU core for packet polling.
- XDP does not need to allocate large pages.
- XDP can work in conjunction with a kernel networking stack.
- XDP does not require special hardware NICs with XDP-supported drivers (however, XDP requires hardware NICs with XDP-supported drivers for higher performance).

2.2 | Related work

2.2.1 | Network monitoring techniques

NetFlow[15] and sFlow (Sample Flow)[16] are two widely used traditional flow monitoring methods. NetFlow captures and aggregates information about flows as they pass through switches. However, NetFlow requires additional memory and workload on a switch CPU to extract and process flow information. In addition, a monitoring system needs to perform polling on switches to obtain report data. Because the time required to export a NetFlow report is relatively long (in the order of ten seconds), NetFlow is not well suited for real-time monitoring. sFlow samples packets from flows passing through switches. sFlow takes a sample once every n packets with a configurable sampling rate $1/n$. Then, the sampled packet is sent immediately to a monitoring system for further analysis. sFlow requires a smaller amount of CPU and memory on switches; however, it has lower accuracy. Sampling methods may be unable to detect small flows, and it can miss network events such as spikes or anomalies.

With the rapid development of SDN technologies, new monitoring methods for SDN environments have recently been introduced. Many of these are based on OpenFlow, the de facto SDN protocol. FlowSense[17] uses OpenFlow *PacketIn* and *FlowRemoved* messages sent from switches to a controller to report network state. FlowSense has a low overhead; however, it cannot provide real-time and accurate network information. OpenNetMon[18] polls switches to get information with an adaptive polling rate. It provides reasonable accuracy while maintaining a low CPU overhead. However, OpenNetMon does not provide information about intermediate switches and lacks a complete view of network state because it polls edge switches only. OpenSample[19] is another monitoring framework that relies on sFlow.

All these systems are constrained because they are based on fixed-function OpenFlow switches. Recently, a programmable data plane has been proposed that changes fixed-function switches to programmable ones. Some new methods that utilize this capability have been proposed, such as OpenSketch,[20] UnivMon,[21] and INT. OpenSketch and UnivMon use 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.

In large and high-speed networks, network monitoring methods mentioned above are unable to provide real-time, fine-grained and end-to-end network information. Moreover, sometimes packet-level monitoring[22,23] is needed, such as when debugging in a multipath routing environment or debugging faulty interfaces that affect only a specific group of packets with the same characteristics. INT has recently been proposed to solve these problems, and it has quickly attracted attention.

2.2.2 | INT solutions and collectors

IntMon[4] focuses on INT implementations in the data plane and the INT controller service in the ONOS controller. A simple collector for INT reports is included in IntMon. However, it is not able to query history of network information because IntMon collector does not store historical data. Moreover, it has a low processing rate and limited scalability because IntMon collector is implemented as an ONOS application.

Prometheus INT exporter[9] is another collector for INT. For every INT report packet, Prometheus INT exporter extracts network information into metrics and pushes the metrics to a gateway. A central Prometheus database server periodically scrapes the latest data from the gateway. Prometheus INT exporter has two problems. First, a high overhead is incurred for processing and sending data to the gateway for every INT report. Second, Prometheus database stores only the latest data from the gateway, that for each scrape. All other data are discarded, although network events, like short traffic bursts, can occur between two scrapes.

Netcope[24] implemented a 100G INT sink on FPGA, to strip and export INT header and data to Flowmon Collector. However, only two types of INT data (Switch ID, timestamp) are allowed and INT data stack length is limited to two.

There are also commercial solutions from Barefoot[25] and Broadcom.[26]

TABLE 1 INT solutions

Feature	INT Functions	Target	Collector	Open Source
IntMon[4]	Src/Sink/Transit	P4 devices	ONOS	O
Prometheus INT Collector[9]	N/A	P4 devices	Prometheus	X
Barefoot DeepInsight[25]	Src/Sink/Transit	P4 devices	DeepInsight	X
NETCOPE 100G INT[24]	Sink	FPGA	Flowmon	X
Broadcom In-band Telemetry in Trident 3[26]	Src/Sink/Transit	Broadcom Trident 3	BroadView Analytics	X
Proposed Architecture	Src/Sink/Transit	P4 devices	INTCollector	O

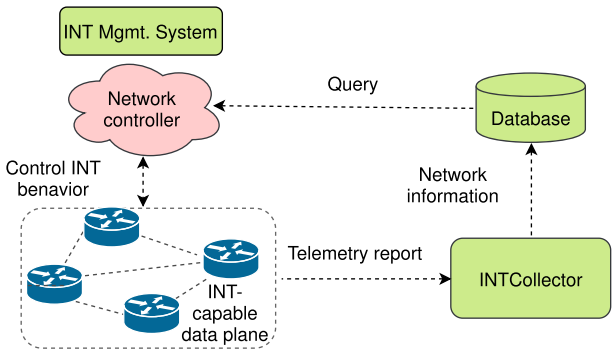


FIGURE 4 INT management architecture

Table 1 summarizes characteristics of INT solutions and the proposed architecture.

3 | ARCHITECTURE DESIGN

In this section, we describe the design of the INT management architecture and its components: INT pipeline, INT management system, and INT collector. As mentioned in the previous section, a common interface for heterogeneous INT-capable switches is defined in the INT management system. The INT collector is designed to lower packet processing overhead and detect network events efficiently.

3.1 | Overview

Figure 4 shows the proposed INT management architecture. It is composed of an INT-capable switches, INTCollector, databases, and a network controller which runs the INT management system. INT data are generated and sent to INTCollector. The collector extracts and filters network information from telemetry reports, converts it into INT metrics and stores those metrics in a database. The network controller controls INT-capable switches and the INT management system controls INT-related behavior in the network. The network controller queries network information from databases and uses the information to control the network.

3.2 | INT-capable data plane

The design of our INT-capable data plane is based on the P4 abstract forwarding model.[10] Regarding the architecture, the processing flow of an INT-capable data plane contains three principal parts: Parser, Ingress, and Egress. The Parser parses packet headers, including INT headers. The Ingress pipeline performs packet forwarding and populates INT metadata for a packet. Finally, the Egress pipeline adds INT data to the packet.

We defined two roles for INT-capable data planes based on the INT specification[5]: *source/sink* and *transit*. A *transit* switch performs INT operations: parsing INT headers and adding INT data specified in the header. A *source/sink* switch includes the capability of a *transit* switch. Additionally, it works as a first hop and last hop switch. A *source/sink* switch adds an INT header to a packet from a host and removes INT header and INT data from a packet that is going to be forwarded to a host. It also generates and sends telemetry report packets[27] to the collector. The INT management system assigns a role to each switch and populates corresponding table entries.

INT metadata are defined as user-defined metadata for INT processing. They consist of *switch_id*, a *source* flag, and a *sink* flag. *switch_id* is assigned by the controller and is fed as one of the INT data items defined in the specification. *source* and *sink* flags are 1-bit fields that indicate if a packet is being forwarded in the first hop switch or last hop switch,

respectively. *mirror_id* field is defined in standard metadata and is used as an identifier of a cloned packet in the data plane. By assigning a specific value to that field, the INT-capable data plane identifies cloned INT packets during INT processing. The behavior is described in detail in the following sections.

3.2.1 | Parser

The main role of the Parser is identifying the existence of INT headers in incoming packets. INT header can be identified by DSCP value in IPv4 header since INT over TCP/UDP encapsulation changes DSCP value to a predefined value. For packets with INT header, the Parser parses INT header and data.

3.2.2 | Ingress pipeline

When packet parsing is completed, packets are fed to the Ingress pipeline for packet forwarding, ie, egress port selection. A basic forwarding Match/Action table (eg, L2 switching and L3 routing) is implemented in the Ingress pipeline. INT processing in the Ingress pipeline is started after a packet passes through forwarding tables so that it can use egress port information. INT sink operation is executed if a host is connected to the egress port.

Algorithm 1 shows processing algorithms in the Ingress pipeline. INT-related packet metadata(*pkt_meta*) are populated after a packet passes through forwarding tables. First, Ingress pipeline determines whether a switch is source (first hop switch in the path) or sink (last hop switch in the path). Then, it sets corresponding flags in the metadata (*source* or *sink*, respectively). Since the control plane has network topology information, it identifies ports that a host is connected to and populates *source_table* and *sink_table* accordingly. As a packet passes those tables, *source* flag is set if packets are from those ports and *sink* flag is set if packets are going to be forwarded to those ports. In addition, a packet is cloned and *mirror_id* in the metadata is populated with the given ID from the controller if the switch is a sink switch. After completing the Ingress pipeline, packets are sent to Queues/Buffers and then sent to the Egress pipeline.

3.2.3 | Egress pipeline

The Egress pipeline adds an INT header and INT data to a packet, mainly because many INT data (eg, hop latency and queue occupancy) become available in Egress. First, Egress checks *source* flag in the packet metadata. If *source* flag is set, it checks whether the packet header matches an entry in watchlist tables, which determines which data packets to monitor by matching packet header fields.[27] If the packet header matches an entry in the table, an INT header is inserted into the packet with parameters sent from the control plane. Second, if an INT header exists in the packet header, INT information of the switch is attached to the end of the INT data. The type of INT data is determined by the value in the instruction field in INT header. Finally, the packet is sent to the Deparser and sent out of the switch.

If a switch is the last hop switch before the destination host, an entire packet with its INT data is cloned. INT data of the sink switch are attached to the cloned packet and are encapsulated within a telemetry report header. The encapsulated packet is forwarded to an external collector. In order to restore an original packet, INT header and INT data are removed from the packet. The packet is then forwarded to the destination. In this way, INT monitoring process is transparent to end hosts.

Algorithm 1 INT processing in Ingress pipeline

Require: *pkt_hdr* - parsed packet header, *mirror_id* - ID of cloned INT packet
Ensure: *pkt_hdr*, *pkt_meta*
  function INGRESS(*pkt_hdr*)
    Apply *basic_forwarding_table* ▷ Set egress port
    if *pkt_hdr* matches *source_table* **then**
      Set *pkt_meta.source* flag ▷ The packet is coming from a host
    end if
    if *pkt_hdr* matches *sink_table* **then**
      Set *pkt_meta.sink* flag ▷ The packet is going to be forwarded to a host
    end if
    if *pkt_meta.sink* **then**
      Clone *pkt* and set *mirror_id* to the cloned *pkt* ▷ Using a pre-defined *mirror_id*
    end if
  end function

Algorithm 2 INT processing in Egress pipeline

```
Require: pkt_hdr, pkt_meta, mirror_id
Ensure: pkt_hdr
  function EGRESS(pkt_hdr, pkt_meta)
    if pkt_meta.source and pkt_hdr matches watchlist_table then
      Add INT header                          ▷ The packet is going to be monitored
    end if
    if pkt_hdr has INT header then
      Add INT data
      if pkt_hdr is cloned and pkt_meta.mirror_id is mirror_id then
        Encapsulate pkt into a telemetry report   ▷ Outer ETH/IP/UDP header is added
      end if
      if pkt_meta.sink then
        Remove INT header and data from pkt_hdr
        Restore original header
      end if
    end if
  end function
```

The Ingress and Egress pipelines are composed of Match/Action tables. Although P4 supports if-else branching, most of the decision logic is implemented with tables for the sake of simplicity. Entries in those tables are populated by the control plane right after the switch is connected to it.

3.3 | INT management system

The INT management system (Figure 5) controls INT-related behavior of INT-capable switches. INT-related behavior includes installing target flows to monitor, specifying types of INT data to collect and configuring the collector information for sink switches. The proposed system is composed of INTIntent, INT Service, an INT driver interface, and a control application.

- INTIntent is a network-level abstraction that carries information for controlling INT-related behavior. With this abstraction, an application can easily populate tables of all INT-capable switches with monitoring rules in the network, without knowing anything about the network it is monitoring, such as network topology or data plane structure of each switch. INTIntent consists of traffic slices (as a five-tuple) and network states to monitor (defined in the INT specification).
- INT Service is an implementation of a pipeline-agnostic northbound API. It orchestrates generation and collection of INT data. The API includes functions for starting/stopping INT, adding/removing INTIntent, and setting/getting INTConfig. INT Service decomposes INTIntent into flow entries and populates those entries in the INT-capable pipelines. It also configures INT-related parameters, such as external collector IP/port. It assigns a role of each INT-capable switch, either *src/sink* or *transit*, and populates flow rules according to the role.
- INT driver interface defines a common interface for managing heterogeneous INT-capable switches. It defines common INT-related behavior that all INT-capable switches should support, such as adding target flows to monitor or configuring the collector IP address and port number. Each INT-capable switch implements the driver interface according to its own data plane structure.
- INT control application is a web GUI. It is used to specify which flows and which network state to monitor. This information is translated into an INTIntent which is then sent to the INT Service.

3.4 | INTCollector

The role of INTCollector is to collect INT data from INT-capable data plane, in the form of telemetry report. It also extracts and filters useful network information from collected data into INT metric values. It then stores those metric values into a database. The SDN controller can query network information from databases and use the information to understand and control the network.

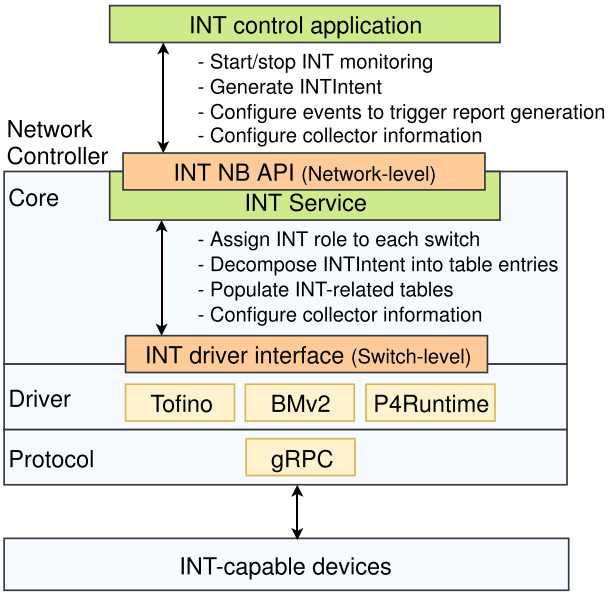


FIGURE 5 INT management system

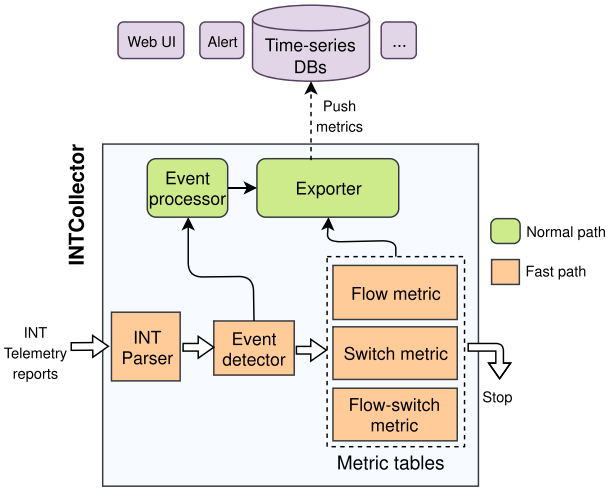


FIGURE 6 INTCollector architecture

Figure 6 presents the design of INTCollector. The rest of this section explains in detail how INTCollector works.

3.4.1 | Metrics

A metric is a data structure to represent network information. Since storing raw INT data is inefficient for processing and querying in databases, the data in telemetry reports are re-organized and defined as a metric with a *metric key* and a *metric value*. A *metric key* is a tuple of *(IDs, measurement)* or *(ids, m)*.

- *IDs*: A tuple of one or several characteristics of flows, networks, or switches that do not change with time (eg, a tuple of switch ID = 2 and egress port ID = 1).
- *measurement*: The type of INT data (eg, switch ID and egress port ID together identify a network link and utilization of this link is a measurement that changes over time).
- *metric value*: the value of a measurement of one metric key at a certain time. For example, hop latency of (sw_id = 4, queue_id = 1) is 1.2 ms at the time point of 10 s.

Metrics can be divided into three types: flow metric, switch metric, and flow-switch metric. Flow metrics include values that are related to flow identification and timestamps. Switch metrics include values that are related to the identification of a switch or switch components and timestamps. Flow-switch metrics include values that are related to the identification of both flow and switch/switch components and timestamps.

TABLE 2 INT metrics

IDs	Measurement	Metric Type
<5-tuple>	Flow path	Flow
<5-tuple>	Flow latency	Flow
<5-tuple + sw_id>	Flow per-hop latency	Flow-switch
<sw_id, queue_id>	Queue occupancy	Switch
<sw_id, queue_id>	Queue congestion	Switch
<sw_id, egress_id>	Link utilization	Switch

The INT specification[5] defines nine fields of INT data: four identification fields (switch ID, ingress port ID, egress port ID, and queue ID), a time field (timestamp), and four measurement fields (hop latency, queue occupancy, queue congestion, and link utilization). From these nine fields, we define six metrics (Table 2).

3.4.2 | Processing flows

INTCollector has two processing paths: a fast path and a normal path. The fast path processes every INT report. Thus, the fast path is required to achieve a high packet processing rate. The normal path processes events sent from the fast path and stores INT metric values in the database.

In the fast path, INT telemetry report packets are passed to the INT parser, which deserializes the packets to extract INT header and INT data. The event detector converts INT data into network metric values and detects network events (Section 3.4.3) by comparing them with the latest values stored in the Info tables. If a network event is detected, it is sent to the event processor in the normal path. Last, metric tables store the latest metric value for each metric key according to the metric type.

In the normal path, the event processor processes network events sent from the fast path. The Exporter gets metric values from two sources: from the event processor in the normal path and from tables in the fast path. It then sends these values to the database.

3.4.3 | Event detection mechanism

The event detector helps to detect network events from INT data. Most of the time, the INT data from several consecutive telemetry report packets will not change significantly (eg, hop latency of a port in a switch may remain the same or change very little over several consecutive telemetry reports). Instead of storing network metrics for each report, the event detector filters important network events to reduce the number of metric values that need to be stored.

We define an event as INT data that contains either a new metric key or a significant change in the value of an existing metric value. Let $\boldsymbol{M}$ be the set of all $(IDs, measurement)$ or (ids, m) in the collector. Let $V_{ids,m}(t)$ be the metric value of (ids, m) at time t. A new event happens when at least one of the following conditions occurs:

- There is a new $(ids, m) \notin \boldsymbol{M}$. For example, a new flow generates events for flow path, flow latency, and flow per-hop latency.
- $\exists (ids, m) \in \boldsymbol{M}$ which satisfies $|V_{ids,m}(t_2) - V_{ids,m}(t_1)| > T(m)$, where t_1 and t_2 are timestamps with $t_2 > t_1$ and $T(m)$ is a threshold for the measurement m. For example, a significant increase in hop latency of (switch 1, port 2) generates an event.

IDs uniquely identifies certain metric value among a set of measurement results with same measurement type.

Using a threshold for event detection significantly reduces the amount of data to be stored in the database, with a trade-off in terms of accuracy. With a smaller threshold T, more accurate metric values can be collected and event detection becomes more sensitive to value changes. With a larger threshold, the number of events and metric values are reduced but the accuracy is also reduced.

3.4.4 | Exporter

The Exporter sends metric values to the database in two different ways: it either periodically pushes the latest values from metric tables or pushes the values when a new event happens. The Exporter pushes metric values into the database periodically for two reasons: to update live status of a metric (especially for flow and flow-switch metrics) and to update the latest value even when there is no network event. Sending data to the database periodically helps in checking live status of a metric.

3.4.5 | Database

The database stores historical INT metric values. A network controller can query network information from the database. The database should support a high write throughput because it is expected that multiple instances of INTCollector will send data to the same database instance. Because INT metrics have their own timestamp and INTCollector needs to push event data, the database should support a custom timestamp and push mechanism.

4 | IMPLEMENTATION

In this section, we describe in detail the implementation of the INT management architecture and INTCollector. We have made our implementation open source.[28]

4.1 | INT management architecture

The proposed INT management architecture is implemented as a part of ONOS, an open-source SDN controller that supports P4.

If an INT-capable device wants to be managed by ONOS, it needs to implement the INT driver interface fitted to its data plane to let ONOS control INT behavior through the unified driver interface. Since implementation details of each INT-capable device are different, the INT driver interface needs to be implemented by pipeline developers. ONOS identifies INT-capable devices by whether a device driver has implemented the INT driver interface or not. When an INT-capable switch is connected to an ONOS instance, it starts initiation process. First, it identifies the role of each device, either source/sink or transit. A switch is identified as a source/sink switch if a host is connected to the switch. Otherwise, the switch is identified as a transit switch. Second, it populates flow tables of the switch with flow rules, which are independent from specific INT intents. This step populates the transit table and source/sink tables. Since the transit table is implemented to match an instruction bit combination on INT header, table entries need to be installed beforehand. For source/sink tables, a packet from a host is marked as an INT source packet by setting an INT source bit in packet metadata. In the same way, a packet that is going to be forwarded to a host is marked as an INT sink packet by setting an INT sink bit in the metadata.

When a collector configuration (eg, collector IP address and port number) is provided by the management application, it is converted into a table entry and installed on all source/sink switches. When an INTIntent is given from the control application, it is converted into table entries to add an INT header to packets that match the given traffic match condition. Removal of INTIntent works in the same way.

4.2 | INTCollector

4.2.1 | Database

The database stores collected INT data. A network controller can query network information from the database. The database should support a high write throughput because it is expected that multiple instances of INTCollector will send data to the same database instance. Because INT metrics have their own timestamp and INTCollector needs to push event data, the database should support a custom timestamp and push mechanism. There are two methods for sending data from the collector to the database: pushing and pulling. Pushing means INTCollector will send data to the database whenever it wants. Pulling means the database decides when to get information by sending a request to INTCollector for data. From the database's view, pulling is easier to implement and more robust. However, a database that supports pushing is more suitable because INTCollector uses event detection.

4.2.2 | Fast path

The INTCollector fast path is implemented in C and accelerated by XDP for higher performance (Figure 7). The fast path XDP program is attached to one or several NICs that receive INT report packets. XDP has a channel to communicate with the normal path in user space.

In our implementation, INTCollector supports IPv4 with INT inside TCP/UDP (Figure 8). An INT telemetry report encapsulates an INT report (inner) inside a UDP packet (outer). The first parser phase deserializes outer header. If the classification does not match, the packet is not a telemetry report and potentially belongs to another application; thus, the packet is passed. In the inner parsing phase, if there is an unmatched classification (which means a packet error), the packet is dropped. The detailed report format can be found in the specification.[5,27]

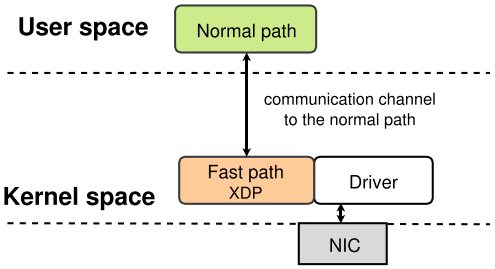


FIGURE 7 INTCollector with XDP

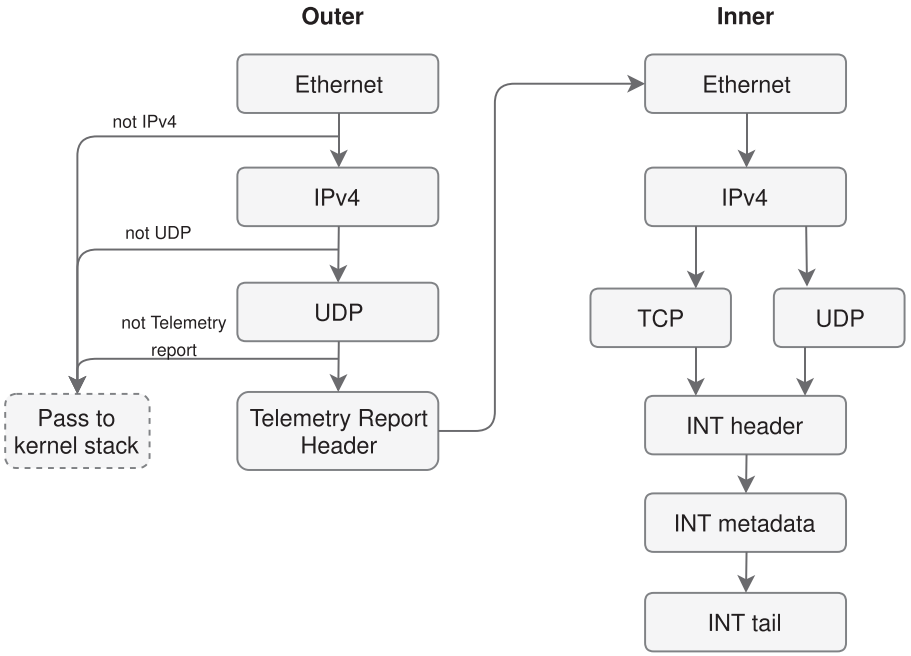


FIGURE 8 INTCollector parser sequence

We use hash tables to store metric values. In these tables, INTCollector stores only the latest values along with a timestamp (for threshold detection, only the last event values with a timestamp are stored). Thus, push metrics are periodically generated by the event detector in the fast path. For other cases, push metric values are read periodically from Info tables in the fast path.

4.2.3 | Normal path

The normal path is implemented in Python for ease of implementation and interaction with the remote database. We used BPF Compiler Collection (BCC)[29] to connect with the fast path and to manage the fast path XDP program. The implementation of the normal path depends on the type of database. As a real-time database, InfluxDB,[30] a high-performance time-series database that supports pushing and custom timestamps, is used since it meets the requirements in Section 3.4.5.

Grafana[31] is used as GUI to access and analyze network metrics.

5 | EVALUATION

In this section, we present the evaluation result of our work. First, we measure the latency in the data plane caused by INT processing. Second, we compare the performance of INTCollector with other collectors and INTCollector itself without event detection. We also investigate how INT report characteristics affect the performance of INTCollector.

5.1 | Processing overhead in the data plane

INT requires additional processing in the data plane, eg, parsing INT headers, matching INT instruction bits, and executing actions to add INT header and INT data. We measured the increased latency in the data plane caused by INT

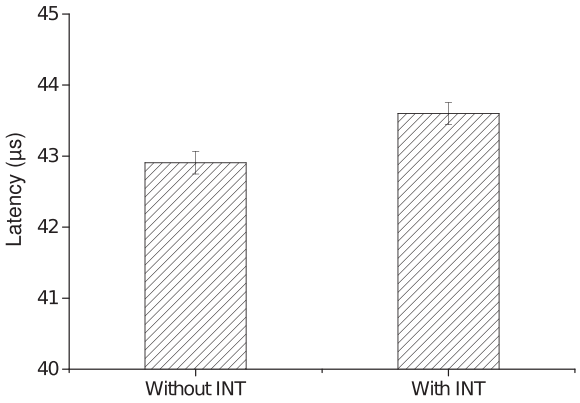


FIGURE 9 Latency in the data plane with and without INT processing

processing. For the evaluation, we have set up a Wedge 100B programmable switch[32] and connected a host. When a switch receives a packet from a host, it sends the packet back to the host. Then, we captured the timestamp of each packet at the host with nanosecond precision, to measure the end-to-end latency. In this way, we could measure the latency accurately, since no time synchronization with nanosecond precision is required. The data plane consists of a basic L2 forwarding table and INT-related tables.

In the first case (without INT), we removed INT-related parsers and tables, so that INT functionalities do not affect the end-to-end latency. In the second case (with INT), we enabled all INT functionalities (source, transit and sink) with basic L2 forwarding tables. Each measurement sent 10 000 packets and calculated the average end-to-end latency of all packets. Figure 9 shows the evaluation result. Enabling INT functionalities in the data plane adds 0.692 microseconds on average, which is 1.587% of the total processing time in the data plane. Since the actual data plane is much more complex (eg, switch.p4 which manipulates traditional switches has more than 35 functions and INT is one of them[33]), the effect of adding INT monitoring capabilities in the data plane is negligible.

5.2 | Collector performance comparison

5.2.1 | Experimental setup

Figure 10 shows the system to evaluate the performance of collectors. Since INTCollector can be considered as a type of Virtual Network Function (VNF) in data center networks, it is reasonable to suppose that INTCollector can be deployed as a virtual machine. Therefore, in the experiement, we set up a VM to measure the performance of INTColletor and other collectors. The host machine is equipped with an Intel Core i5 3570 CPU (3.4 GHz) and 12 GB DDR3 RAM. It runs Ubuntu 18.04 64-bit with kernel v4.15. Each virtual machine (VM) has one vCPU core with 2 GB RAM and runs Ubuntu 18.04 64-bit with kernel v4.15. The VMs are accelerated by a kernel-based virtual machine (KVM).

The tests were conducted as follows. INT telemetry report packets are sent to VM1 over a TAP interface.[34] INT reports contain only new-key events and no significantly changed events. We developed a dedicated Python program to generate custom INT reports that suit the purpose of each test. Then the collector (INTCollector, IntMon Collector, or Prometheus INT exporter) receives and processes INT report packets in VM1. Extracted metric values are written to the database server, which is hosted in VM2. We measured average CPU usage of VM1 over 3 minutes.

We used vhost-net and virtio-net (with multi-queue enabled) to accelerate the network at the host and guest sides respectively. We also used a huge page setup for VM1-KVM. However, virtio-net is still a bottleneck because it starts to drop packets when the report packet rate is increased to around 1.2 Mpps. Therefore, we set up another system with

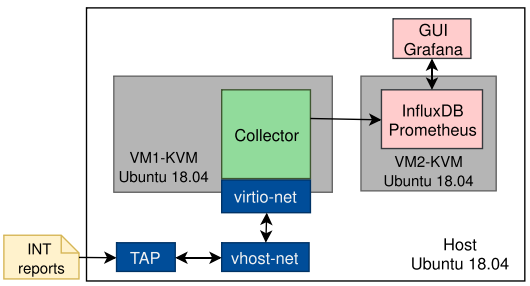


FIGURE 10 Experiment setup

high-throughput hardware NICs to overcome the bottleneck. The system also adopts SR-IOV (Single Root I/O Virtualization) to further enhance the INTCollector's performance, in terms of throughput and CPU usage.[35] Section 5.4 presents the evaluation results.

5.2.2 | INTCollector vs other collectors

We compared IntMon collector, Prometheus INT Exporter, and our INTCollector in VM1 by sending an INT report to VM1. The INT report has one flow with six path hops. A total of nine INT fields are collected. For IntMon collector, there is no data to send to VM2 because it does not have a database component.[4] For Prometheus INT Exporter, there is a gateway between Prometheus INT Exporter and Prometheus server. We put the gateway in VM2.

In all cases, CPU usage increased linearly with the report packet rate (20% fixed CPU usage for Prometheus INT exporter). However, there are huge differences in the CPU usage efficiency, as shown in Figure 11, note that both axes use a log-scale. IntMon Collector has the worst performance and INTCollector outperforms the others.

IntMon collector has the lowest performance. For an additional 1% of CPU usage, INT report rate increases by 0.1 kpps (kilo packets per second; linear regression: $C = 9.71 \times R + 1.64$, where C is CPU usage in % and R is the report rate in kpps). IntMon collector has a very high overhead because it is implemented as an ONOS application; thus, packets need to pass through ONOS before entering the collector.

Prometheus INT Exporter has better throughput. For an additional 1% of CPU usage, INT report rate increases by 5.7 kpps ($C = 0.175 \times R + 22.9$). The rate of increase is 57 times higher than that of IntMon collector. Prometheus INT exporter is implemented in Java as a stand-alone application. Since it is separated from ONOS, it is faster than IntMon collector. However, Prometheus INT exporter has a high static CPU cost of about 20%.

INTCollector has the best throughput out of the three. For an additional 1% of CPU usage, INT report rate increases by 154.8 kpps ($C = 0.00646 \times R + 0.082$). Hence, the rate of increase is 27 times faster than that of Prometheus INT Exporter. INTCollector uses event detection which helps reduce CPU usage and storage costs. XDP helps improve the processing throughput of the fast path.

There are other implementations such as EverFlow,[22] that also collect packet-level telemetry for network monitoring, but they are not specific to INT. The throughput of Everflow is 4.8 Mpps with a 16-core Intel Xeon CPU running at 2.1 GHz (0.3 Mpps per core, but core utilization is unknown), which is several times lower than that of INTCollector.

5.2.3 | INTCollector with or without event detection

We measured the CPU usage of INTCollector with event detection disabled (Figure 11). This means that the fast path sends all INT data to the normal path and INT metric values of all INT reports will be sent to databases. When event detection is disabled, CPU usage increases linearly with input report rate. However, for a 1% increase in CPU usage, report rate only increases by 0.19 kpps ($C = 5.39 \times R + 3.69$), which is more than 800 times lower than INTCollector with event detection. Without event detection, the overhead of processing all INT reports in the normal path (which is written in Python) becomes too high and the throughput of INTCollector is reduced significantly.

5.3 | Effect of INT characteristics on CPU usage

We measured how INT report characteristics affect the performance of INTCollector. Considered characteristics include the number of flows, the number of hops in INT reports, selection of INT fields, and the frequency of network events. The same environment as in previous section is used for the experiment. In general, CPU usage increases gradually as the amount of information in INT reports increases (Figure 12). In the first test, we increased the number of flows. INT report consists of six hops and collects switch IDs only. As the number of flows increased, the CPU usage increased to 8.79% for

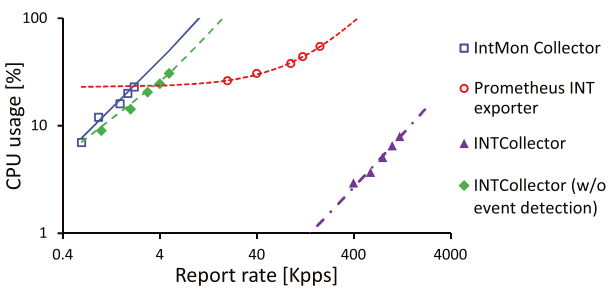


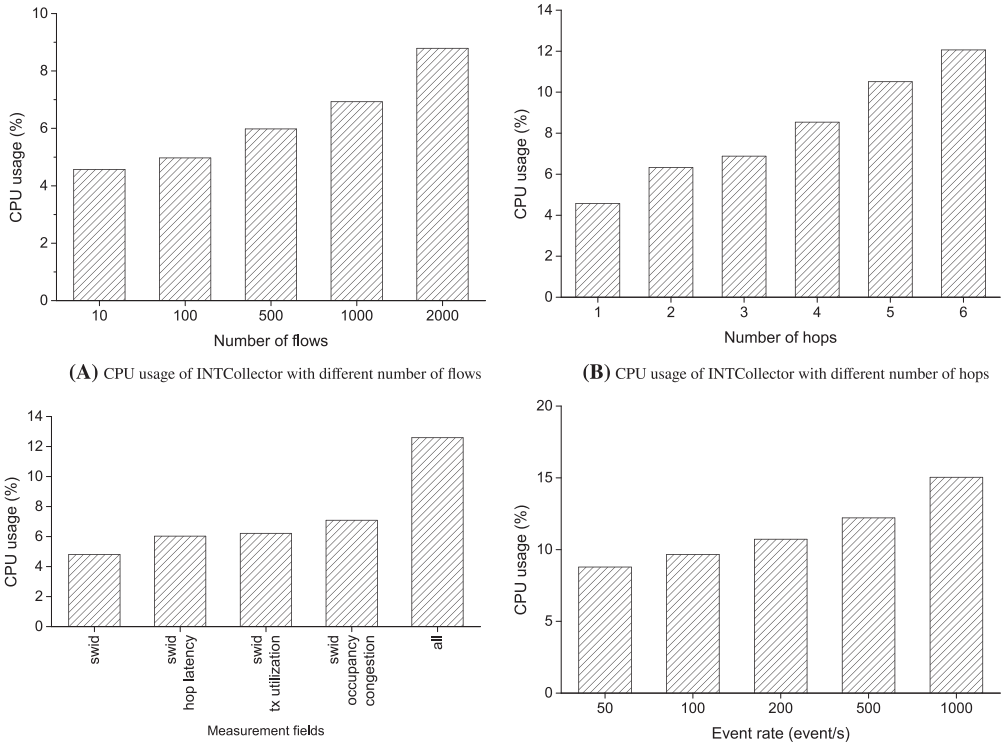
FIGURE 11 CPU efficiency of IntMon Collector, Prometheus INT Exporter, INTCollector, and INTCollector without event detection

2000 flows. In the second test, we increased the number of hops in the INT reports, which were generated from 100 flows. All nine INT fields were collected. As the number of hops increased, the CPU usage increased to 12.06% for six hops. In the third test, we changed INT fields that were collected. The INT reports were generated from 100 flows and each report has six hops of INT data. While the type of INT field does not affect CPU usage, the number of collected INT fields does. The CPU usage increased to 12.59% when all INT fields were collected. In the last test, we increased the network event rate. The INT reports were generated from 100 flows and each report has three hops of INT data. All nine INT fields were collected. As the event rate increased, the CPU usage increased to 15.04% with 1000 event/s.

5.4 | Performance with hardware NIC and SR-IOV

We set up another system which is equipped with four CPU cores and a hardware NIC (Figure 13). The system also adopts SR-IOV, to overcome the limitation mentioned in Section 5.2.1 and further enhance the INTCollector's performance. We firstly disabled SR-IOV in Host 2 and measured average CPU usage of the host with different report traffic rate. Then, we enabled SR-IOV in the host and ran the measurement again to see the impact of SR-IOV. We did the same measurement five times and calculated the average in each case.

Figure 14 shows the results. In all cases, INTCollector consumes less CPU resources when SR-IOV is enabled. In case of low report rate (eg, 2 Gbps), the CPU usage is reduced 49.38% by enabling SR-IOV. When the report rate is increased, the CPU usage gap is decreased (6.15% with 10 Gbps report rate). We can also see rapid increment in CPU usage when the report rate is increased to 6 Gbps. One possible reason is that the system cannot process reports fast enough with excessive



(A) CPU usage of INTCollector with different number of flows

(B) CPU usage of INTCollector with different number of hops

(C) CPU usage of INTCollector with different number of measurement fields

(D) CPU usage of INTCollector with different event rates

FIGURE 12 Effects of INT characteristics on CPU usage

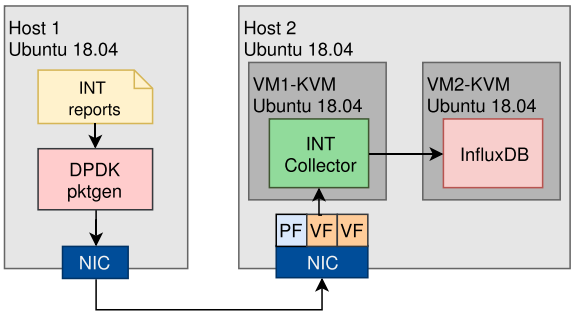


FIGURE 13 Experiment setup for hardware NIC and SR-IOV

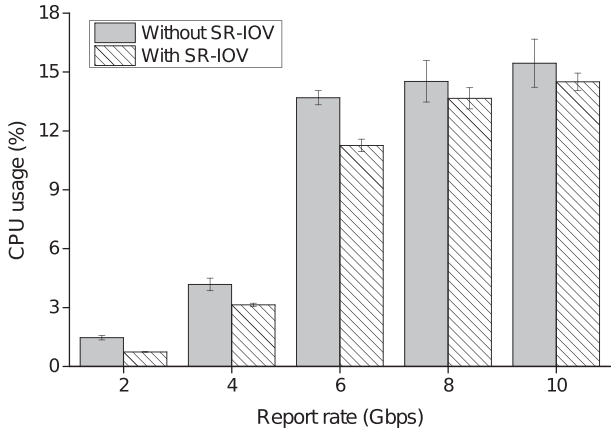


FIGURE 14 The effect of SR-IOV to the CPU Usage of INTCollector

incoming traffic. Each XDP process reports right after a packet is received at the ring buffer of the NIC in softirq phase. If the system cannot process fast enough, it queues remaining report packets into ksoftirqd[36] to process later, which causes a rapid increase in CPU usage for the ksoftirqd process.

6 | DISCUSSION

6.1 | Limitations

Our current implementation of INTCollector has several limitations. First, the maximum number of hops is limited to six. In the fast path, we need to use a loop to parse INT data. However, the size of an XDP program is limited and loops in XDP programs need to be unrolled to ensure that the code terminates. As the allowed number of hops is increased, unrolled XDP program size also increases. The maximum size of an XDP program is exceeded when we set the number of hops to seven. Although maximum number of hops is limited to six, this is enough for most data centers and 5G edge computing environments where measuring network latency is important, since popular network topologies for data centers, such as two-tier, three-tier, and fat-tree, do not need more than six hops for end-to-end packet delivery. Second, if some events need to be detected and solved in strict real time (eg, detecting loops), INTCollector may be delayed because it pushes events to InfluxDB first and then the network controller queries the information from InfluxDB. One solution is to report these events directly to the controller.

6.2 | Future issues

In this section, we discuss future issues and unsolved problems in this work.

6.2.1 | Placement of INTCollector

In the experiment, we assumed that INTCollector is connected directly to sink switches, so that telemetry report traffic is transmitted through a dedicated channel to the collector. However, in the real world, it is infeasible to place collectors on every sink switch although it does need to be placed somewhere in the network. In that case, telemetry report traffic to the collector should be treated the same as data traffic. To minimize the impact of report traffic, an optimal way to place collector instances is required (eg, by minimizing the number of hops from all sink switches).

6.2.2 | Fast path in the data plane

It is possible to implement a data plane function to extract INT data and filter network events in sink switches using a programmable data plane such as P4.[10] This approach ensures processing at line rate. However, there are several problems with this approach: switch resources are limited and need to be shared with other functions; the telemetry report specification uses UDP to send an INT report packet from the switch to the collector,[27] which results in the loss of important information; and it is hard to monitor live status of the metrics. Another feasible approach is adopting FPGA-based P4-enabled SmartNIC to offload the parsing logic. It can reduce overhead in kernel space.

6.2.3 | Side effects of event detection

Although the event detection mechanism helps to reduce the number of report packets that are processed in the collector, it affects the accuracy of INT-based measurement since report packets without an event are discarded. We need to investigate the effects of event detection on the accuracy of the measurement. More specifically, the trade-off between threshold value for the event detection and the accuracy needs to be investigated. For network monitoring purpose, the threshold value can be defined based on the network administration policy. For anomaly detection, it needs to be tuned adaptively to optimize the accuracy and the number of anomalies detected.

6.3 | Use cases

In this section, we discuss two use cases for the proposed architecture: traffic engineering (TE) and accurate end-to-end latency monitoring in a 5G environment.

6.3.1 | Traffic engineering

Current TE algorithms in SDN (eg, Hedera[37] and MicroTE[38]) suffer from a large overhead and huge computation costs, which need additional hardware or can be applied only to large flows. Moreover, an SDN controller would become overloaded while collecting flow information from switches in the network.

Using the proposed architecture, fine-grained traffic information can be collected and analyzed in the management plane, which reduces the overhead for the controller. Moreover, the traffic information is refined and clustered in the management plane, which reduces the number of values input to TE algorithms.

By merging the centralized network view from the SDN controller, various traffic characteristics can be analyzed, such as traffic size, interval, duration, and end points. TE algorithms make use of this information, eg, by changing the flow path to minimize flow latency. This information is also useful for forecasting short-term and long-term traffic status, which enables effective TE.

6.3.2 | Accurate end-to-end latency measurement in a 5G environment

A 5G mobile network enables various mission-critical services, which are not possible in current 4G environments[39] because 5G supports enhanced mobile broadband and ultra-reliable low-latency communication. Each service has its own required latency and data rate, which are specified in the service level agreement (SLA) between each service provider and network operator. Network operators need to measure the SLA parameters of each service to ensure that they meet the SLA. The proposed architecture can be used to measure end-to-end latency of each service accurately and detect SLA violations when they happen. Moreover, Multi-access Edge Computing (MEC) in 5G environment requires extremely low latency, therefore finding the root cause of delayed packet would be necessary in MEC and it would be one of the killer use case of the proposed approach.

7 | CONCLUSIONS

In this paper, we have presented the design and implementation of an INT management architecture and its two main components: INT management system and INTCollector. INT management system is designed to control heterogeneous INT-capable devices and INTCollector is a high-performance collector that collects INT report packets from those devices. We defined INT network metrics to represent network information. We also proposed a mechanism to filter network events from a huge amount of telemetry report packets with those metrics. The event detection mechanism helps extract only important network information to store in databases.

In data centers where 10G, 40G, and 100G links are common, telemetry report rate can be very high, so we may need a system with multiple collectors. Because INTCollector instances can work independently, INTCollector can be scaled by adding new instances. Our work can be used for automated network operation and management with the help of artificial intelligence (AI) technologies. As future work, we are planning to work on methods to distribute telemetry reports to collector nodes depending on the processing capability of each node and to implement event detection in the data plane. We are also planning to develop methods to apply our monitoring system to various scenarios including 5G environment and service function chaining.

ACKNOWLEDGEMENTS

This work was supported by the ICT R&D program of MSIT/IITP (2018-0-00749, Development of virtual network management technology based on artificial intelligence).

ORCID

Jonghwan Hyun https://orcid.org/0000-0003-2134-1173

REFERENCES

1. Mizrahi T, Sprecher N, Bellagamba E, Weingarten Y. An overview of operations, administration, and maintenance (OAM) tools. Organization: InternetEngineering Task Force (IETF). RFC 7276; 2014.
2. Kim C, Sivaraman A, Katta N, Bas A, Dixit A, Wobker LJ. In-band network telemetry via programmable dataplanes. In: Demo paper at ACM SIGCOMM. Santa Clara, CA; 2015.
3. Song H, Zhou T, Li Z, et al. Toward a network telemetry framework. Internet draft, IETF; https://tools.ietf.org/html/draft-song-opsawg-ntf-02; 2018.
4. Tu NV, Hyun J, Hong JWK. Towards ONOS-based SDN monitoring using in-band network telemetry. In: 19th Asia-Pacific Network Operations and Management Symposium (APNOMS). IEEE; 2017; Seoul, Korea:76-81.
5. The P4.org Applications Working Group. In-band Network Telemetry (INT) specification v1.0. https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf
6. Berde P, Gerola M, Hart J, et al. ONOS: towards an open, distributed SDN OS. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN). ACM; 2014; New York, NY, USA:1-6.
7. The P4 Language Consortium. The P4 language specification 1.1.0. https://p4lang.github.io/p4-spec/docs/P4-16-v1.1.0-spec.pdf
8. Tom H, Alexei S. eXpress Data Path (XDP) programmable and high performance networking data path. https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf
9. Serkant U. Prometheus INT exporter. https://github.com/serkantul/prometheus_int_exporter
10. Bosshart P, Varghese G, Walker D, et al. P4: programming protocol-independent packet processors. *ACM SIGCOMM Comput Commun Rev*. 2014;44(3):87-95.
11. Netcope. 100G In-band Network Telemetry With Netcope P4. https://www.netcope.com/getattachment/670aabd2-89f6-4ecf-8620-9b437a256f24/100G-In-band-Network-Telemetry-With-NP4.aspx
12. P4 Runtime—a control plane framework and tools for the P4 programming language. https://github.com/p4lang/PI
13. Brenden B. eXpress Data Path: Getting Linux to 20 Mpps. Linux Meetup Santa Clara; 2016.
14. Intel. Data Plane Development Kit (DPDK). https://dpdk.org
15. Claise B. Cisco systems NetFlow services export version 9. RFC 3954, IETF; 2004. https://www.ietf.org/rfc/rfc3954.txt
16. Wang M, Li B, Li Z. sFlow: Towards resource-efficient and agile service federation in service overlay networks. In: Proceedings of the 24th International Conference on Distributed Computing Systems. IEEE; 2004; Tokyo, Japan:628-635.
17. Yu C, Lumezanu C, Zhang Y, Singh V, Jiang G, Madhyastha HV. FlowSense: monitoring network utilization with zero measurement cost. In: International Conference on Passive and Active Network Measurement (PAM). Springer; 2013; Hong Kong:31-41.
18. van Adrichem NLM, Doerr C, Kuipers FA. Opennetmon: network monitoring in openflow software-defined networks. In: 2014 IEEE Network Operations and Management Symposium (NOMS). IEEE; 2014; Krakow, Poland:1-8.
19. Junho S, Kwon TT, Dixon C, Felter W, Carter J. Opensample: a low-latency, sampling-based measurement platform for commodity SDN. In: 2014 IEEE 34th International Conference on Distributed Computing Systems. IEEE; 2014; Madrid, Spain:228-237.
20. Yu M, Jose L, Miao R. Software defined traffic measurement with opensketch. In: Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13). ACM; 2013; Berkeley, CA, USA:29-42. http://dl.acm.org/citation.cfm?id=2482626.2482631
21. Liu Z, Manousis A, Vorsanger G, Sekar V, Braverman V. One sketch to rule them all: rethinking network flow monitoring with univmon. In: Proceedings of the 2016 ACM SIGCOMM Conference. ACM; 2016; New York, NY, USA:101-114. http://doi.acm.org/10.1145/2934872.2934906
22. Zhu Y, Kang N, Cao J, et al. Packet-level telemetry in large datacenter networks. *ACM SIGCOMM Comput Commun Rev*. 2015;45(4):479-491. http://doi.acm.org/10.1145/2829988.2787483
23. Jeyakumar V, Alizadeh M, Geng Y, Kim C, Mazières D. Millions of little minions: using packets for low latency network programming and visibility. *ACM SIGCOMM Comput Commun Rev*. 2015;44(4):3-14.
24. Benáček P, Puš V, Kekely M, Richter L, Minařík P, Pazdera J. 100G In-Band Network Telemetry with P4 and FPGA. In: The 4th P4 Workshop. Stanford, CA; 2017.
25. Barefoot deep insight. https://www.barefootnetworks.com/products/brief-deep-insight/
26. Broadcom Trident 3 In-band Telemetry. https://people.ucsc.edu/~warner/Bufs/Trident3-telemetry.pdf
27. The P4.org Applications Working Group. Telemetry report format specification v1.0. https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report.pdf

28. In-band Network Telemetry (INT) with ONOS and P4. https://wiki.onosproject.org/display/ONOS/In-band+Network+Telemetry+(INT)+with+ONOS+and+P4
29. BPF compiler collection (BCC). https://github.com/iovisor/bcc
30. InfluxDB: Scalable datastore for metrics, events, and real-time analytics. https://github.com/influxdata/influxdb
31. Grafana: the open platform for beautiful analytics and monitoring. https://grafana.com
32. WEDGE 100BF-32X 100GBE DATA CENTER SWITCH. https://www.edge-core.com/productsInfo.php?id=335
33. Switch.p4. https://github.com/p4lang/switch
34. Universal TUN/TAP device driver. https://www.kernel.org/doc/Documentation/networking/tuntap.txt
35. Dong Y, Yang X, Li J, Liao G, Tian K, Guan H. High performance network virtualization with SR-IOV. *J Parallel Distrib Comput*. 2012;72(11):1471-1480. https://doi.org/10.1016/j.jpdc.2012.01.020
36. Software interrupt context: softirqs and tasklets. https://www.kernel.org/doc/htmldocs/kernel-hacking/basics-softirqs.html
37. Al-Fares M, Radhakrishnan S, Raghavan B, Huang N, Vahdat A. Hedera: dynamic flow scheduling for data center networks. In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)ACM; 2010; San Jose, CA, USA:19-19. http://dl.acm.org/citation.cfm?id=1855711.1855730
38. Benson T, Anand A, Akella A, Zhang M. Microte: fine grained traffic engineering for data centers. In: Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies (CoNEXT). ACM; 2011; New York, NY, USA:8:1-8:12.
39. Parvez I, Rahmati A, Guvenc I, Sarwat AI, Dai H. A survey on low latency towards 5g: ran, core network and caching solutions. *IEEE Commun Surv Tutor*. 2018;20(4):3098-3130.

AUTHOR BIOGRAPHIES

Jonghwan Hyun is a PhD student in the Department of Computer Science and Engineering at Pohang University of Science and Technology (POSTECH). He also received his BS degree in the Department of Computer Science and Engineering from POSTECH in 2011. He has worked as a research scholar at Open Networking Foundation (ONF) where he drove a project on In-band Network Telemetry Service and open sourced it through ONOS. His research interests include P4, programmable network, Software-Defined Network (SDN), Network Function Virtualization (NFV), and network traffic monitoring and analysis.

Nguyen Van Tu is a Research Assistant in the POSTECH Information Research Laboratories at Pohang University of Science and Technology (POSTECH). He received his BSc degree in Electronic and Telecommunication from Hanoi University of Science and Technology (HUST) in 2015 and MSc degree in Computer Science and Engineering from POSTECH in 2018. His research interests include Software-Defined Network (SDN), Network Function Virtualization (NFV), and programmable network.

Jae-Hyoung Yoo is a Research Professor in the Graduate School of Information Technology at Pohang University of Science and Technology since 2013. He received his BSc and MSc degrees in Electrical Engineering from Yonsei University, Korea, in 1983 and 1985, respectively, and the PhD degree in Computer Engineering from Yonsei University, Korea, in 1999. He had worked for KT (Korea Telecom) from 1986 to 2012, where he was responsible for developing operations and management systems for various network systems, such as PSTN, ATM, and Internet. He was a Network PM (Program Manager) of Ministry of Science, ICT and Future Planning in Korea from 2016.1 to 2018.3. He has been an active volunteer in various committees in IEEE NOMS, APNOMS, and KICS KNOM. His research area includes network management and security, software-defined networking (SDN), and machine learning-based virtual network and system management.

James Won-Ki Hong is Professor in the Department of Computer Science and Engineering at Pohang University of Science and Technology (POSTECH). He had worked as the Chief Technology Officer and Senior Executive Vice President for KT (Korea Telecom), the largest telecommunications company in Korea from March 2012 to February 2014, where he was responsible for leading the R&D effort of KT and its subsidiary companies. He was Chairman of National Intelligence Communication Enterprise Association and Chairman of ICT Standardization Committee in Korea. He cofounded and is currently Executive Director of SDN/NFV Forum in Korea. His research interests include network innovation, such as software-defined networking and network function virtualization, cloud computing, mobile services, IPTV, ICT convergence technologies (eg, Smart Home, Smart Energy, and Health care), and Internet of Things. James had served as the Head of Department of Computer Science and Engineering,

Dean of Graduate School of Information Technology, Director of POSTECH Information Research Labs, and Head of the Division of IT Convergence Engineering at POSTECH. James received his HBSc and MSc degrees in Computer Science from the University of Western Ontario, Canada, in 1983 and 1985, respectively, and the PhD degree in Computer Science from the University of Waterloo, Canada, in 1991.

How to cite this article: Hyun J, Nguyen VT, Yoo J-H, Hong JW-K. Real-time and fine-grained network monitoring using in-band network telemetry. *Int J Network Mgmt*. 2019;29:e2080. https://doi.org/10.1002/nem.2080