Master's Thesis

# INTCollector: A High-performance Collector for In-band Network Telemetry

Nguyen Van Tu

Department of Computer Science and Engineering

Pohang University of Science and Technology

2018

# INTCollector: In-band Network Telemetry를 위한 고성능 수집기

# INTCollector: A High-performance Collector for In-band Network Telemetry

# INTCollector: A High-performance Collector for In-band Network Telemetry

by

Nguyen Van Tu
Department of Computer Science and Engineering
Pohang University of Science and Technology

A thesis submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of Master of Science in the Computer Science and Engineering

Pohang, Korea
6. 15. 2018
Approved by
James Won-Ki Hong (Signature)
Academic advisor

# INTCollector: A High-performance Collector for In-band Network Telemetry

Nguyen Van Tu

The undersigned have examined this thesis and hereby certify that it is worthy of acceptance for a master's degree from POSTECH

6. 15. 2018

| | | |
|---|---|---|
| Committee Chair | James Won-Ki Hong | (Seal) |
| Member | Young-Joo Suh | (Seal) |
| Member | Hanjun Kim | (Seal) |

MCSE Nguyen Van Tu
20162115
INTCollector: A High-performance Collector for In-band Network Telemetry,
INTCollector: In-band Network Telemetry를 위한 고성능 수집기
Department of Computer Science and Engineering , 2018, 36p, Advisor : James Won-Ki Hong. Text in English.

# ABSTRACT

In Software-Defined Networking (SDN), monitoring is an essential function to provide information about the network and help the SDN controller to make network controlling decisions. In-band Network Telemetry (INT) is a new method that can provide real-time, fine-grained, and end-to-end network monitoring. INT works by embedding network information into every packet. At the final switch in the path, INT extracts network information into report packets and sends the reports to a collector. However, the huge amount of data from INT requires high processing capability of the collector.

We present the design and implementation of INTCollector, a high performance collector for INT. We propose a mechanism to extract important network information, called event, from INT raw data. The mechanism filters network events, reduces the number of metric values that need to be stored, reduces CPU

usage and storage cost while still ensuring to capture and store all important network information. INTCollector has two processing flows: a fast path to process INT report packets, and a normal path to process events and store metric values into a database. The fast path is accelerated by eXpress Data Path (XDP) - a Linux in-kernel fast packet processing framework. Our calculation shows that event detection can massively reduce the amount of data need to be stored (two to three orders of magnitude in our test scenario). The evaluation shows that INTCollector can process INT reports at the rate of 1.2 Mpps with 8% of CPU when running with software NIC. We expect better result can be achieved with XDP supported hardware NICs.

# Contents

# List of Tables

# List of Figures

# I. Introduction

Monitoring is an important function in any networking systems, especially in Software-Defined Networking (SDN) [1]. In SDN, the control plane and the data plane are separated: the data plane forwards the packets between nodes; the control plane controls the data plane through a standard south-bound protocol, such as OpenFlow [2]. The control plane is usually a logically centralized SDN controller. To control the network efficiently and ensure that the network operates smoothly in an optimal way, the controller needs to have information about the network state. This information (e.g., link utilization) is essential for the controller to make suitable decisions (e.g., traffic engineering). Monitoring also helps to troubleshoot the network problems, such as failures in links or switches. Monitoring thus becomes an essential function in any SDN networking systems. Furthermore, the information needs to be real-time, fine-grained, and provide global visibility of the network.

Among many available methods to monitor an SDN network, In-band Network Telemetry (INT) [3] has many advantages. INT attaches network information (e.g., switch ID, hop latency, or link utilization) to every packet at every switch in the path. At the final switch before reaching the destination (sink switch in INT), the INT telemetry report packet, which carries the 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. Also, INT works

directly at the data plane and does not require the involvement of the control plane at the running phase. However, packet-level monitoring can produce very high report rate (the number of INT report packets sent to the collector). Even with techniques to reduce the number of reports, such as monitoring only targeted flows, the report rate can easily get to millions of packet per second [4]. Thus, there is a need for a high-performance collector to process INT telemetry reports.

In this thesis, we present the design and implementation of INTCollector - a high-performance collector for INT. We define INT metrics of network information, propose a mechanism to extract important network information, which we called event, from INT telemetry reports. The information is then stored in a time-series database. The mechanism filters only network events (such as new flow, or significant change of hop latency), thus helps reduce CPU usage and storage cost, while still guaranteeing that the important information is captured and stored. From our calculation, INTCollector can reduce the amount of data need to be stored by two or three orders of magnitude. We split INTCollector into two processing paths: a fast path to process INT report packets, and a normal path to process events and store data into a database. We then accelerate the fast path with eXpress Data Path (XDP) [5] - an in-kernel fast packet processing framework. In our evaluation, we run INTCollector in a Virtual Machine (VM) and send INT reports to the VM via virtio-net software NIC. INTCollector can process INT report packets at the rate of 1.2 Mpps with 8% of CPU usage. We expect that INTCollector can achieve better throughput with an XDP-supported

hardware NICs.

The remainder of the thesis is organized as follows. In Section II, we provide literature of network monitoring techniques, discuss techniques for fast packet processing, and discuss related work about collectors for INT. In Section III, we present the detailed design of INTCollector, including the design overview, processing flow, metrics, network event detection mechanism, and the database side. Section IV presents the implementation details of INTCollector. Section V shows the evaluation results of INTCollector in various aspects. Finally, Section VI concludes this thesis.

# II. Background and Related Work

In this section, we survey various techniques to monitor a network and show that INT is superior to the other methods. Then, we explain how XDP work and how it can improve the performance of INTCollector. Finally, we survey related work about collectors for INT.

## 2.1 Network monitoring

NetFlow [6] and SFlow [7] are two traditional methods that have been widely used for many years. NetFlow captures the information of the flows when they pass through switches, and then aggregate the data. NetFlow requires additional memory and workload on the switch CPU to extract and process flow information. In addition, to obtain report data, a monitoring center needs to perform polling on the switches. Because the period for exporting NetFlow data is long (e.g., 15 s), NetFlow is not well suited for real-time monitoring. Sample Flow (SFlow) samples the packet flows passing through the switches. SFlow takes a sample once every $n$ packets, with the configurable sampling rate $n$. The sampled packet can then be sent immediately to a monitoring probe 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 events, such as spikes or anomalies, or be unable to detect small flows.

With the rapid development of SDN, new monitoring methods for SDN en-

vironment have recently been introduced. Many of these are based on OpenFlow, the de-facto SDN protocol. FlowSense [8] uses OpenFlow *PacketIn* and *FlowRemoved* messages sent from switches to a controller to report the network state. FlowSense has low overhead, but it is not able to do real-time and accurate monitoring. OpenNetMon [9] polls the switches to get information, with an adaptive polling rate: the rate increases when flow rate changes rapidly, and decreases when they stabilize to minimize the number of queries. Adaptive polling provides reasonable accuracy while maintaining low CPU overhead. However, because OpenNetMon polls only the edge switches, it does not provide information regarding intermediate switches and lacks a complete view of the network state. OpenSample [10] is another monitoring framework that use packet sampling. OpenSample's use cases focus on traffic engineering because it can quickly detect elephant flows and estimate link utilization.

In large and high-speed networks, these above monitoring systems are unable to provide real-time, fine-grained, and end-to-end network information. In-band Network Telemetry is recently proposed to solve this problem, and quickly gains attention.

## 2.2 In-band Network Telemetry

In-band Network Telemetry (INT) [3] is a monitoring technique designed to collect and report network states directly from the data plane. In every switch in the flow path, INT attaches the information to the packets that pass through the switches (Fig. 2.1). At the final switch (sink switch), the information is extracted,

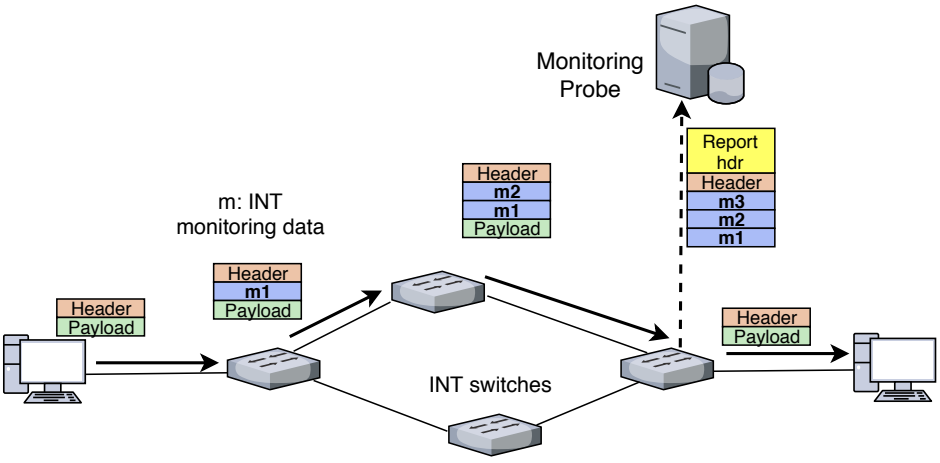


Figure 2.1: INT working process

encapsulated inside a report packet called telemetry report [11], and then sent to a collector. The information can be any data provided by the switches, such as timestamp, hop latency, or link utilization. INT parameters, such as which information to collect, are included in the INT header as "telemetry instruction". Usually, the telemetry instructions are controlled by a central controller. The structure of INT report packet and other information of INT can be found in the specification [12].

Compared to the monitoring methods mentioned above, INT can provide real-time, fine-grained, and end-to-end network monitoring. INT enables many advanced applications such as network troubleshooting, advanced congestion control, advanced routing, or network data plane verification [12]. INT performs all the monitoring function in the data plane, thus the network with INT enabled should still be able to run at full line-rate. Usually, INT is implemented with programmable data plane, which means network functions (switches) can be re-

programmed. Several INT implementations have been done recently [3, 13, 14].

## 2.3 Collectors for INT

IntMon [13] provides a collector for INT reports. However, IntMon focuses on INT implementation in the data plane and INT controller services in Open Networking Operating System (ONOS) [15] controller, thus IntMon collector is quite simple and lacks necessary functions. IntMon collector does not store historical data, so we cannot query the history of network information later. Also, IntMon collector is implemented as an ONOS application, thus has high overhead and cannot process INT reports in high rate (as will be shown in the evaluation).

Prometheus INT exporter [16] is another collector for INT. For every INT report packet, Prometheus INT exporter extracts network information into metrics and pushes the metric values to a gateway. A central Prometheus database server then scrapes the latest data from the gateway periodically.

Prometheus INT exporter has two problems. Firstly, it has high overhead of processing and sending the data to the gateway for every INT report. Secondly, even though the gateway helps to temporary store short-lived information, Prometheus database only stores latest data from the gateways for each scrape, thus may lose network events.

## 2.4 eXpress Data Path

eXpress Data Path (XDP) [5] is a kernel framework that allows doing packet processing inside the kernel. Unlike kernel modules, which can affect the kernel

stability, XDP is safe and secure for the kernel. To achieve high throughput, XDP programs immediately process the packets right after they arrive at the NIC, thus avoid the cost of kernel networking stack processing and kernel-user space switching. XDP can get to 10 Mpps forwarding rate and 20 Mpps for packet dropping [17].

Beside XDP, there are also other frameworks for fast packet processing, such as DPDK [18]. However, compared to DPDK, XDP has several advantages: XDP does not requires dedicated CPU for packet polling, so we have additional CPU resource for INT processing; XDP does not require allocating large pages; XDP can work in conjunction with the kernel networking stack; and XDP does not require special hardware NICs with supported drivers (however, XDP requires hardware NICs with XDP-supported drivers for higher performance).

XDP has limitations. To ensure the safety of the kernel, XDP programs have restricted programming capability. For example, the total number of instructions is limited. Thus, XDP should only be used for tasks that are simple but require to be done fast. We solve this problem by partitioning INTCollector into a normal path and a fast path, with the fast path can be run as an XDP program (Section III).

# III. Design

## 3.1 INTCollector architecture

INTCollector is designed as one component in a complete SDN system, as shown in Fig. 3.1. The SDN controller controls the behaviors of the network, such as packet forwarding. The network needs to support INT monitoring, and INT data are sent to INTCollector in the form of telemetry report. The collector processes INT data, extracts and filters useful network information into INT metric values, then stores them into a database. Finally, the SDN controller can query network information from the database, and use the information to understand and control the network. We have developed an actual system [19] like Fig. 3.1 has shown, but the details is out of the scope of this work.

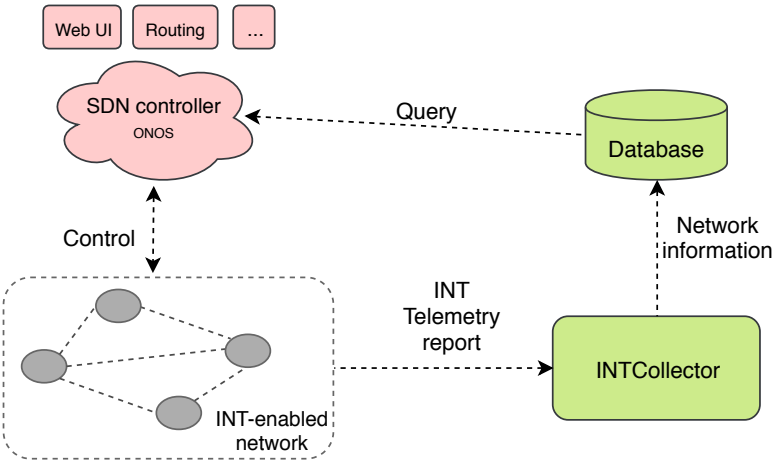


Figure 3.1: SDN network with INTCollector

The architecture of INTCollector is presented in Fig. 3.2. The rest of this chapter explains how INTCollector works in detail.

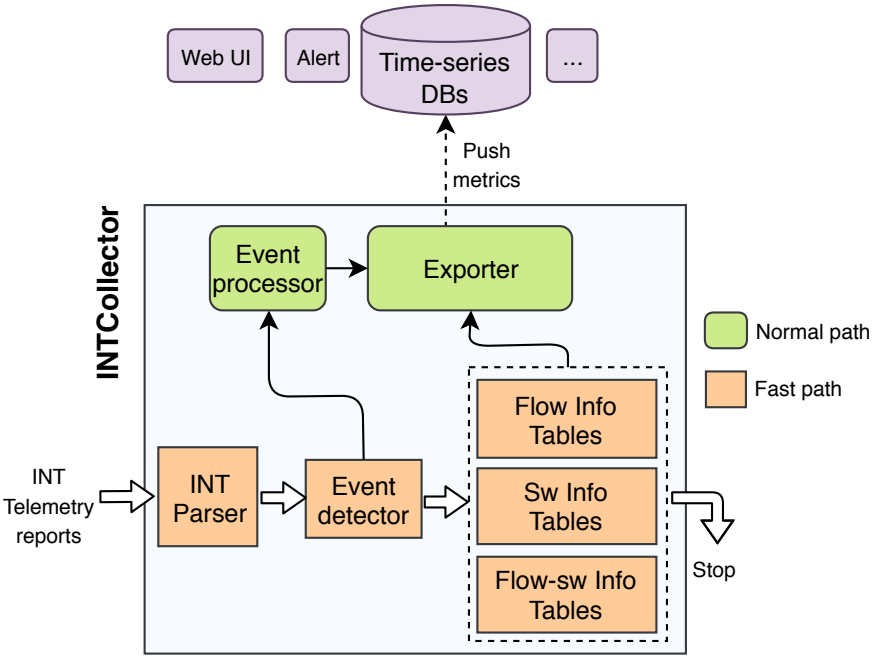


Figure 3.2: INTCollector architecture

## 3.2 Processing flows

INTCollector has two processing paths: a normal path and a fast path. The fast path processes INT reports and needs to run for every new INT telemetry report. Thus, the fast path needs to be done very quickly. The normal path processes events sent from the fast path, and stores INT metric values to the database. The normal path also needs to be done efficiently, but not as urgent as the fast path.

In the fast path, INT telemetry report packets from INT data plane are first

passed to INT parser. INT parser deserializes the packet structure to get INT header and data. Next, Event detector converts INT data into network metric values, and detects the network events (Section 3.4) by comparing with the last values stored in the Info tables. The network events (if any) are then sent to the normal path. Finally, the latest metric values are stored in the Info tables (Section 3.3).

In the normal path, Event processor processes network events that are sent from the fast path, then sends the metric values to Exporter. Exporter gets metric values from Event processor and the tables from the fast path, then sends these values to the database, which can be the local database or remote database.

We split INTCollector into the fast path and the normal path for easy implementation and optimization. The fast path could be implemented in a low-level language (such as C language), and we can apply some acceleration techniques for the fast path (such as XDP in our implementation). The normal path can be written in a high-level language (such as Python) for easy processing and interacting with the remote database.

## 3.3 Metrics

Metrics represent network information. We cannot just store INT raw data, which is defined for doing INT function in the data plane, not for easy processing and querying in a database. Instead, we need to convert INT data into more meaningful metrics. In INTCollector, we define a metric with *metric key* and *metric value*. A *metric key* is a tuple of *(IDs, measurement)*, or *(ids, m)*. The

*IDs* is a tuple of one or several characteristics of flows, networks or switches that do not change with time (e.g., a tuple of switch ID = 2 and egress port ID = 1). The *measurement* is the measurement that we want to know and its value changes with time (e.g., switch ID and egress port ID identify a network link, and the utilization of this link is a measurement that changes over time). A *metric value* is the value of a measurement of one metric key at one time point. For example, hop latency of (sw_id = 4, queue_id = 1) is 1.2 ms at the time point of 10 s.

The metrics can be divided into three types:

- **Flow metrics**: the metric with the value depending on the flow identification and the timestamp. For example, flow path, and end-to-end flow latency can be classified as flow metrics.
- **Switch metrics**: the metric with the value depending on the identification of the switch or switch components, and the timestamp. For example, queue occupancy, queue congestion, and link utilization are switch metrics.
- **Flow-switch metrics**: the metric with the value depending on the identification of both the flow and the switch/switch components, and the timestamp. For example, flow per-hop latency can be classified as a flow-switch metric.

INT specification [12] define total nine fields of INT data: four identification fields (switch ID, ingress-egress port ID, and queue ID), one time field (timestamp), and four measurements (hop latency, queue occupancy, queue congestion,

and link utilization). From these nine fields, we extract six metrics, as shown in Table 3.1. Note that 5-tuple is the combination of source / destination IPs, source / destination ports, and protocol.

Table 3.1: INT metrics

| **IDs** | **Measurement** |
|---|---|
| <5-tuple> | Flow path |
| | Flow latency |
| <5-tuple + sw_id> | Flow per-hop latency |
| <sw_id, queue_id> | Queue occupancy |
| | Queue congestion |
| <sw_id, egress_id> | Link utilization |

## 3.4 Event detection mechanism

Event detector helps detect network events from INT reports. INT produces very fine-grained network information (packet-level granularity, the information is attached to every packet and reported to the collector). Most of the time, the data of several consecutive INT report packets can be the same (e.g., hop latency of a port in a switch may keep the same or change very little in several consecutive INT reports). In most case, it is unnecessary to store network metrics for every INT reports. We usually care about important information changes in the network, such as new flow, path change, hop latency increases too much (which may indicate congestion). Event detection filters only important network events, helps reduce the number of metric values that need to be stored, thus reduces the

storage cost and the CPU usage in both INTCollector and the database server.

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

- 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.
- $\exists (ids, m) \in \boldsymbol{M}$ so that new $V_{ids,m}(t_2)$ changes significantly compared to $V_{ids,m}(t_1)$, with $t_2 > t_1$. Usually, $t_2$ is the timestamp of the new INT report, and $t_1$ is the old timestamp. For example, hop latency of (switch 1, port 2) increases significantly, which generates an event.

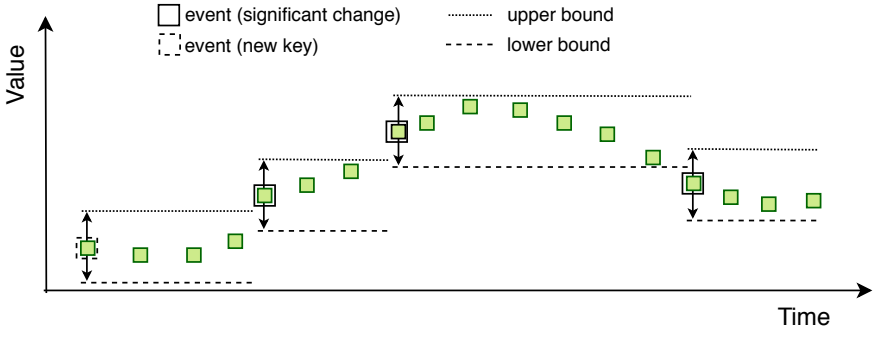


Figure 3.3: New events when metric value changes significantly

To define "change significantly", we compare the difference between the new value and the last value. If the difference is higher than a threshold, a new event occurs. Let $T(m)$ be the threshold for the measurement $m$. An event is generated

when $|V_{ids,m}(t_2) - V_{ids,m}(t_1)| > T(m)$, with $t_2 > t_1$, and $|x - y|$ is the absolute value of $x - y$. Fig. 3.3 shows an example of events generated when the metric value changed. The first event is new key event, and the next three are significant change events.

Using threshold for event detection can massively reduce the amount of data to be stored in the database, with the accuracy trade-off. With a smaller threshold $T$, we have more accurate metric values and can detect more value changes, but at the cost of more events and a higher number of metric values to be stored. With a bigger threshold, we can reduce the number of events and metric values, but the measurement is less accurate.

The point of event detection is to monitor and store not all network information, but only changes that are important. For example, with hop latency, sudden changes may indicate micro traffic bursts, which should be captured; gradually increase or decrease may indicate the change of traffic throughput, which should also be captured; but small fluctuations usually have no significant meaning and can be safely (also should be) ignored. Also, with threshold detection method, even without any new events, we still know the upper bound and lower bound of the metric value (e.g., when the last hop latency is 130, threshold is 40, and there is no new data, we still know that either the current hop latency is between $130 \pm 40$), or the metric does not exist anymore. These two situations can be differentiated by periodically pushing metric values (Section 3.5.1).

There is another way to define "change significantly" by using interval. We can partition the value space of $V_{ids,m}(t)$ into interval $i$. For example, if the value

space is partitioned equally with the interval period of 100, then we have interval 0-100, 100-200, 200-300, and so on. Call $I_{ids,m}(t)$ the interval of the metric value of $(ids, m)$ at time point $t$, then a significantly-changed event is generated when $I_{ids,m}(t_1) \neq I_{ids,m}(t_2)$, with $t_1 > t_2$. Using interval also helps reduce the amount of data need to be stored in the database; and when there is no new event, we still know the interval of the metric values. However, compared to using threshold, using interval has a disadvantage: a small change happens in the boundary of two consecutive intervals may result in an event with not much actual value change (e.g., from 99 to 101). Thus, interval event detection should only be used if we need to know when the value passes a fixed boundary.

## 3.5 Exporter and Database

### 3.5.1 Exporter

Exporter establishes the connection with the database and sends the metric values to the database periodically or when a new event happens. With the help of Event detection, we can relax the time period for sending metric values to the database.

Exporter updates metric values to the database periodically because of two reasons: to update the live status of metrics (especially for flow and flow-switch metrics), and to update the newest value even when there is no network event. When there is no event of a metric sent to the database, we do not know whether the metric is still existing or not. Periodically sending data to the database can help to check the live status of the metric.

### 3.5.2 Database selection

The database is where we store historical INT metric values. SDN controller can query network information from the database. The database should support high write throughput because we expect multiple instances of INTCollector can 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 custom timestamp and push mechanism. There are two methods for sending data from the collector to the database: pushing, and pulling. Pushing means the INTCollector can push the data to the database whenever it wants. Pulling means the database decides when to get the information by sending a request to INTCollector to retrieve the data. From the database's view, pulling method is easier to implement and more robust. However, because INTCollector uses event detection, a pushing-supported database is more suitable.

Table 3.2: InfluxDB and Prometheus

| **InfluxDB** | **Prometheus** |
|---|---|
| - Time-series databases<br>- High performance (500k sample/s for InfluxDB, 800k sample/s for Prometheus)<br>- Support various client programming languages<br>- Rich extensions (UI, alert, etc.) | |
| - Support event push, custom timestamp | - No custom timestamp, limited event push |
| - Complex query, higher storage | - Flexible query, lower storage |

With all the above needs, we chose InfluxDB [20]. InfluxDB is a high-performance time-series database which supports pushing and custom times-

tamp. InfluxDB also provides rich extensions such as Alert and GUI. Prometheus INT exporter, on the other hand, uses Prometheus [21] for their INT Collector. Prometheus provides higher performance than InfluxDB with a more flexible query language, but does not support custom timestamp and has limited pushing (Prometheus use pulling method). Table 3.2 summaries InfluxDB and Prometheus.

# IV. Implementation

## 4.1 Fast path

The INTCollector fast path is implemented in C language, and accelerated by XDP for higher performance (Fig. 4.1). The fast path XDP program is attached to one or several NICs that receive INT report packets. XDP also has a fast channel for communication with the user space, where we run the normal path. XDP requires Linux kernel v4.8 or later. Using XDP for the fast path has a limitation: the maximum number of hops is 6 in our current implementation. In the fast path, we need to use loop to parse INT data. However, current XDP verifier (which is used to ensure that the XDP program is safe for the kernel) requires the XDP program to un-roll the loop to prevent unsafe code, which results in increasing the program size. XDP program size is limited, thus the number of hops is limited. The looping limitation was already aware by the Linux kernel developers and there are incoming patches to allow loop [22].

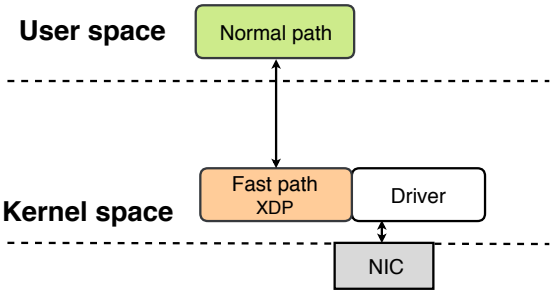


Figure 4.1: INTCollector with XDP

There is another option to implement the fast path: in the sink switch. It is possible to implement a data plane function to extract INT data and filter network events in the switches by using programmable data plane such as P4 [23]. This approach can produce the highest throughput and ensure the line-rate speed. However, there are several problems with this approach: the in-switch resources are limited and need to be shared with other functions; the switch - collector connection protocol is UDP [11], which can cause loss of important information; it is hard to upgrade the algorithm because of the resource constraint; and it is hard to monitor the live-status of the metrics.

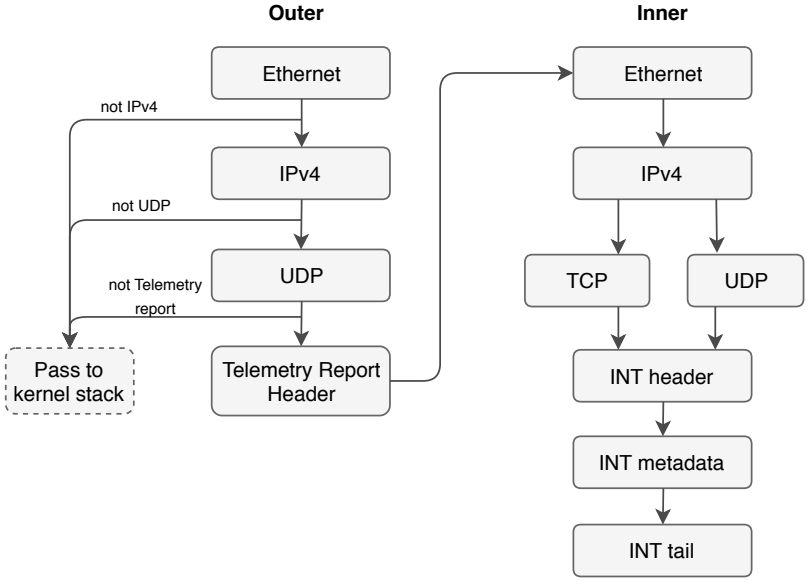


Figure 4.2: INTCollector parser sequence

In our current implementation, INTCollector only supports IPv4 with INT inside TCP/UDP (Fig. 4.2). INT telemetry report encapsulates INT report (Inner) inside a UDP packet (Outer). The first parser phase deserializes the

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 any unmatched classification (which means packet error), the packet is dropped. The detailed report format can be found in the specification [12, 11]. INTCollector supports both specification v0.5 and v1.0 for INT and telemetry report.

## 4.2 Normal path

The normal path is implemented in Python for easy implementation and interaction with the remote database. We use BPF Compiler Collection (BCC) [24] to connect with the fast path and manage fast path XDP program. The implementation of the normal path also depends on the selection of the database.

```
{
      "measurement": "queue_occupancy,sw_id={0},queue_id={1}".format(
                                  sw_id, q_id),
       "time": timestamp,
       "fields": {
                  "value": queue_occupancy,
        }
}
```

Figure 4.3: InfluxDB example for queue occupancy

For the database, INTCollector has main support for InfluxDB, because of the reasons explained in Section 3.5.2. Because InfluxDB only supports one dimension (timestamp) data, we need to combine the metric *key* and *measurement* (Fig. 4.3). We use Grafana [25] as the GUI to access and analyze the network metrics. INTCollector also has partial support for Prometheus, but without the event detection, which can result in losing important network information.

# V. Evaluation

In this chapter, we first evaluate the accuracy and efficiency of event detection (Section 5.2). Then, we compare the performance of INTCollector versus other collectors and versus itself when disabling event detection (Section 5.3). Next, we see how INT report characteristics (the number of metric values, the number of hop in the flow path, the frequency of network events, and the selection of INT fields) affects INTCollector (Section 5.4). Finally, we compare INTCollector performance when using InfluxDB and Prometheus (Section 5.5).

## 5.1 Experiment setup

We set up the system to evaluate INTCollector as shown in Fig. 5.1. The host machine uses Intel core I5 3570 CPU, 12GB DDR3 RAM and runs Ubuntu 18.04 64 bit with kernel v4.15. Each Virtual Machine (VM) has one vCPU core with 2GB RAM, and also runs Ubuntu 18.04 64 bit with kernel v4.15. VMs are accelerated by Kernel-based Virtual Machine (KVM).

Tests are mostly conducted as follow (any differences will be mentioned in the details of each test). Firstly, INT telemetry report packets, which are stored in the INT report file, are sent to the VM1 over the TAP interface by using a packet replay tool such as Tcpreplay [26]. INT reports contain only new-key events and no significantly-changed event (except the event rate test in Section 5.4). We developed a dedicated Python program to generate custom INT reports

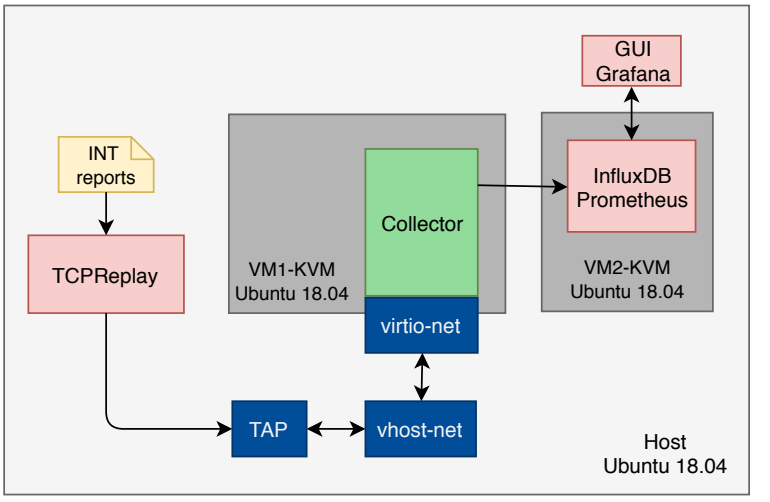


Figure 5.1: Experiment setup

that suit the purpose of each test. Then, the collector (INTCollector, IntMon Collector, or Prometheus INT exporter) listens to INT report packets in VM1, processes the reports, then writes the metric values to the database server, which is hosted in VM2. For INTCollector, the database server is InfluxDB, except the test in Section 5.5. We measure the average CPU usage in three minutes of the VM1. In addition, we can use GUI tool (such as Grafana) to query and analyze the metric values from the database.

In our test, we use vhost-net and virtio-net (with multi-queue enabled) to accelerate the network in the host and guest sides, respectively. We also use huge page setup for VM1-KVM. Even with that, Tcpreplay and virtio-net are still the bottleneck points. We expect to set up better test with high throughput hardware NICs in the future work.

## 5.2 Event detection

### 5.2.1 Effect of threshold value

For this test, we compare the hop latency values detected by INTCollector with the known hop latency from INT report packets. We use both low report rate (10 pps) and high report rate (1 Mpps) and get the same results. We use two values of threshold: 40 ns and 20 ns (Fig. 5.2).

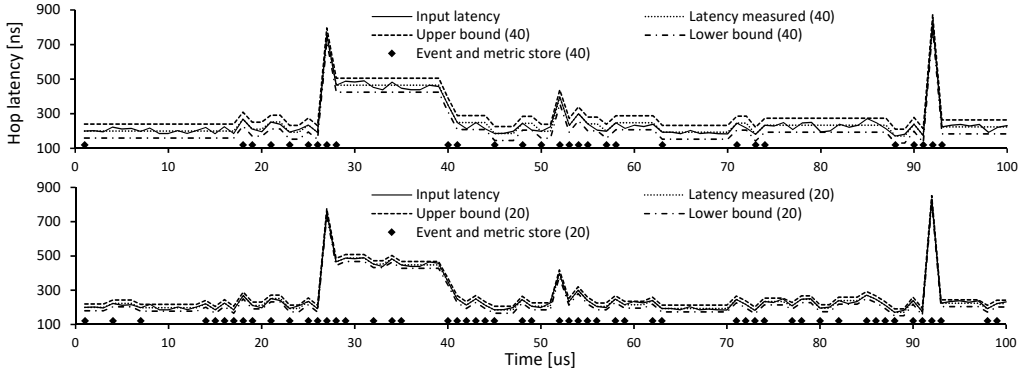


Figure 5.2: Event detection for hop latency with different threshold

In both cases, the event detection works as expected. When the event happens, INTCollector provides the actual hop latency. During the period when the event does not occur, INTCollector guarantees that the hop latency is always between the upper bound line and lower bound line. When the threshold is 40 ns, Mean Square Error (MSE) of the measured latency is 16.0 ns, and the number of events is 29. When the threshold is 20 ns, we have more accurate latency: MSE is 7.3 ns, smaller changes are also detected; but the number of event is higher (54 events), which means higher CPU usage and higher number of metric values to

store.

### 5.2.2 Number of metric values

We calculate the number of metric values that need to be stored. Let $R$ be the average INT report rate (in Mpps), $k$ the average metric values in one INT report, $K$ the total number of metric keys, $T$ the monitoring time (in second), and $E$ the average number of events per second. In the naive approach, where all INT report data are stored, the number of metric values is $M_{all} = R \times T \times k \times 10^6$. In INTCollector, the number of metric values is $M_{event} = (E + \frac{K}{P}) \times T$, where $P$ is the periodically push period (in second).

To easily see how many metric values need to be stored for each case, we consider a fat-tree network with 20 switches, each switch has 4 ports; there are 5000 flows which are spread all the network, the average flow path is 5 hops; all 9 INT fields are monitored, and the INT report rate $R$ is 1 Mpps. Then, the number of metric values for each INT report is $k = 22$ (1 for flow path, 1 for flow latency, 5 for each of flow per-hop latency, queue occupancy, queue congestion, and link utilization), the total number of metric keys is $K = 110240$ ( 5000 for flow path, 5000 for flow latency, 100000 for flow per-hop latency, 80 for each of queue occupancy, queue congestion, and link utilization). Assuming the periodically push period is $P = 10$ s, we have:

$$\frac{M_{all}}{M_{event}} = \frac{1 \times 22 \times T \times 10^6}{100 + \frac{110240}{10} \times T} = \frac{22000000}{E + 11024}$$

Assume that $E = 1000$, then $\frac{M_{all}}{M_{event}} = 1830$, which means event detection reduces the number of metric values by more than 1800 times. With $E = 10000$,

event detection reduces the number of metric values by 1000 times. Even with $E = 50000$, which we consider very high, event detection still reduce the number of metric values by 360 times.

## 5.3 Performance comparison

### 5.3.1 CPU usage

For this test, we run IntMon collector, Prometheus INT Exporter, and our INTCollector in VM1; and send INT report to VM1 in different rates. INT report has 1 flow with 6 hops in the path and total 9 INT fields are activated. For IntMon collector, there is no data to send to VM2, and Tcpreplay is replaced by a simulated network data plane with INT function to generate INT reports [13]. For Prometheus INT Exporter, there is a gateway between Prometheus INT exporter and Prometheus server. We put the gateway in VM2.

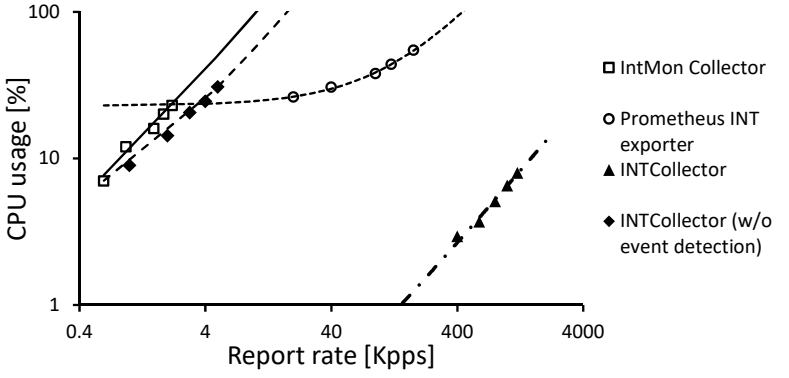


Figure 5.3: CPU efficiency of IntMon Collector, Prometheus INT exporter, INTCollector, and INTCollector w/o event detection

In all cases, the CPU usage increases linearly with the report packet rate. However, there are huge differences in the CPU usage efficiency, which are shown

in Fig. 5.3 in log-scale. IntMon Collector has the worst performance, and INT-Collector outperforms the other two.

IntMon Collector has the lowest performance. It can process only 0.1 Kpps (Kilo packets per second) of INT report rate with additional 1% CPU usage (linear regression: $C = 9.71 \times R + 1.64$, where $C$ is CPU usage (%) and $R$ is Report rate (Kpps)). IntMon collector is implemented as an ONOS application, thus has very high overhead because the packets need to pass through ONOS before entering the collector.

Prometheus INT exporter has better throughput. It can process 5.7 Kpps of INT report rate with additional 1% of CPU ($C = 0.175 \times R + 22.9$), which is 57 times faster than IntMon collector. Prometheus INT exporter is implemented in Java as a stand-alone application, separated from ONOS, thus is faster than IntMon collector. Prometheus INT exporter seems to have a high static CPU cost at about 20%.

INTCollector has the best throughput out of three. INTCollector can process INT report at the rate of 154.8 Kpps with additional 1% of CPU usage ($C = 0.00646 \times R + 0.082$), which is 27 times faster than Prometheus INT exporter. INTCollector uses event detection which helps reduce CPU usage and storage cost; and XDP which helps improve the processing throughput of the fast path.

There are other implementations which also do packet-level telemetry for network monitoring, but not INT-specific, such as Everflow [4]. Everflow can process 4.8 Mpps with 16 cores Intel Xeon CPU with 2.1 GHz speed (0.3 Mpps per core, but the core utilization is unknown), which is several times lower than

INTCollector.

**Storage cost and delay**

**In term of long-term storage cost**, IntMon Collector does not store historical data to any database. Prometheus INT exporter sends INT metric values of all INT reports to a remote gateway, so the number of metric values sent to the remote gateway is similar to the $M_{all}$, which is calculated in Section 5.2.2. However, Prometheus server only collects the latest data from the gateway periodically, which reduces the amount of data stored, at the cost of possibly losing network events. For INTCollector, the number of metric values sent to database is $M_{event}$, which is calculated in Section 5.2.2.

**In term of the delay in updating metric values**, IntMon Collector process INT reports and update latest metric values to local in-memory tables, thus there is almost no delay. Prometheus INT exporter pushes metric values to the gateway, then the gateway needs to wait for the Prometheus database to scrape the data, thus the delay is high (can be equal to the pulling period, which is usually set to few seconds). INTCollector pushes metric values directly to the database without waiting the pulling period, thus there is no significant delay.

**In term of in-memory cost**, Prometheus INT exporter do not store metric values in RAM. IntMon Collector and INTCollector store latest metric values in in-memory hash tables, which could handle millions of entries. For example, to store flow latency, INTCollector uses a hash table with 13-byte key size (the size of 5-tuple) and 8-byte value size (32 bits for latency value, and 32 bit for latest timestamp). Thus, 21 MB RAM can store 1 million entries for flow latency.

### 5.3.2 INTCollector when enable and disable event detection

We also measure the CPU usage of INTCollector when event detection is disabled (Fig. 5.3) (which means the fast path sends all INT data to the normal path and INT metric values of all INT reports will be sent to the database). When event detection is disabled, the CPU usage also increases linearly with the input report rate, but INTCollector can only process 0.19 Kpps of report rate for 1% of CPU ($C = 5.39 \times R + 3.69$), which is more than 800 times slower than INTCollector with event detection enabled. Without event detection, the overhead of processing all INT reports in the normal path (which is written in Python) becomes too high and massively reduce the throughput of INTCollector.

## 5.4 Effect of INT characteristics to CPU usage

We measure the CPU usage in case of INT report characteristics (the number of hop in the flow path, the number of metric values, the frequency of network events, and the selection of INT fields) changed (Fig. 5.4). The INT report rate is fixed to 1 Mpps.

In general, CPU usage increases gradually when the amount of information in INT report increases. In the first test, we increase the number of hops. INT report has 100 flows and all INT fields are activated. CPU usage increases with the number of hops, gets up to 12.06% at 6 hops. In the second test, we increase the number of metric values by increasing the number of flows. INT report has 6 hops with only switch ID is activated in INT. CPU usage increases when the number of flows increases, gets up to 8.79% at 2000 flows. In the third test, we

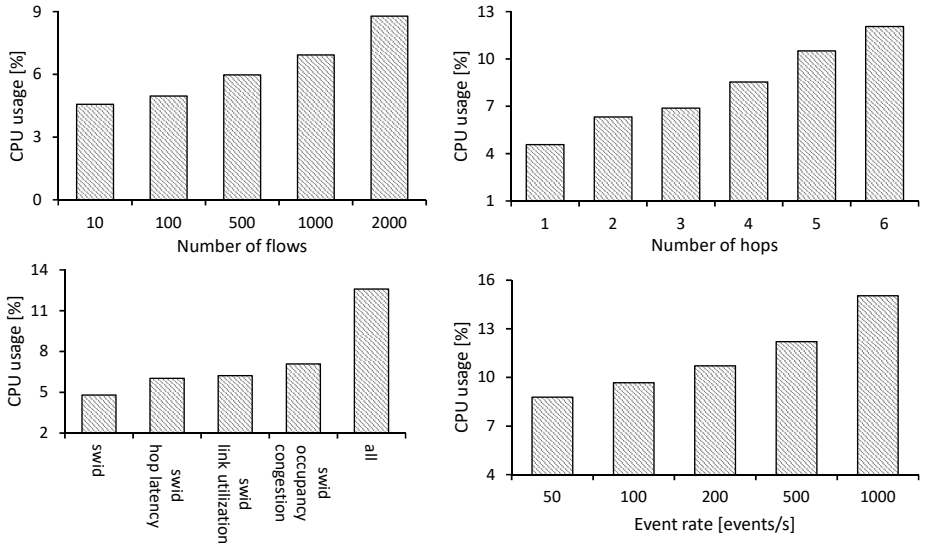


Figure 5.4: INT characteristics affect CPU usage

change the INT fields that are activated. INT report has 100 flows and 6 hops. CPU usage increases when the number of activated INT fields increases, gets up to 12.59% when all INT fields are activated. In the final test, we increase the network event rate. INT report has 3 hops, 100 flows with all 9 INT fields are activated. CPU usage increases with the event rate, gets up to 15.04% with 1000 event/s.

## 5.5 INTCollector with InfluxDB and Prometheus

In this test, we measure the CPU usage of INTCollector with InfluxDB and Prometheus when changing INT report packet rate (Fig. 5.5. INT report has 1 flow with 6 hops and 9 INT fields. The period to periodically send data metric values is set to 5 s for Prometheus and 10 s for InfluxDB (because event detection

does not work with Prometheus, the period is set to be smaller than InfluxDB). The CPU usage increases with the report rate, with the CPU usage when using Prometheus was slightly lower than when using InfluxDB. The reason is InfluxDB client processing consumes more CPU than Prometheus client processing. In general, we can get 1.2 Mpps of INT report rate with less than 8% of CPU usage.

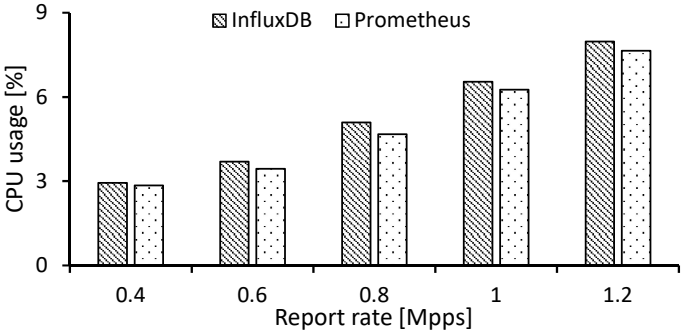


Figure 5.5: CPU usage of INTCollector when using InfluxDB and Prometheus

# VI. Conclusion

INT is a new network monitoring technique with many advantages: real-time, end-to-end monitoring, packet level granularity of information. However, the huge amount of report packets from INT requires a high-performance collector to process them.

In this thesis, we presented the design and implementation of INTCollector, a high-performance collector for In-band Network Telemetry. We defined INT network metrics to represent network information. With these metrics, we proposed a mechanism to filter network events from the huge amount of INT report packets. The event detection mechanism helps extract only important network information to store in the database, thus reduces the CPU usage in both collector and database, and reduces the storage cost. INTCollector is divided into the fast path and the normal path. The fast path then is accelerated by XDP for higher performance.

In data centers where 10G and 100G links are common, INT report rate can be very high so that we may need multiple collectors. We plan to develop a system with multiple instances of INTCollector to process INT reports and send INT metric values to a common InfluxDB instance. We will need to develop an algorithm to distribute INT reports to each collector node depend on the processing capability of each node. The application can be implemented on top of an SDN controller, such as ONOS controller.

# References


[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[2] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[3] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2015.

[4] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 479–491, New York, NY, USA, 2015. ACM.

[5] Herbert Tom and Starovoitov Alexei. eXpress Data Path (XDP) Programmable and high performance networking data path. [Online] `https://`

github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf.

[6] Benoit Claise. Cisco systems NetFlow services export version 9. *RFC 3954 (Informational), Internet Engineering Task Force*, 2004.

[7] Mea Wang, Baochun Li, and Zongpeng Li. sFlow: Towards resource-efficient and agile service federation in service overlay networks. *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 628–635, 2004.

[8] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V Madhyastha. FlowSense: Monitoring network utilization with zero measurement cost. *International Conference on Passive and Active Network Measurement*, pages 31–41, 2013.

[9] 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*, pages 1–8, 2014.

[10] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. OpenSample: A low-latency, sampling-based measurement platform for commodity SDN. *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 228–237, 2014.

[11] The P4.org Applications Working Group. Telemetry Report Format Specification v0.5 and 1.0. [Online] https://github.com/p4lang/p4-applications/blob/master/docs.

[12] The P4.org Applications Working Group. In-band Network Telemetry (INT) specification v0.5 and 1.0. [Online] https://github.com/p4lang/p4-applications/blob/master/docs.

[13] N. Van Tu, J. Hyun, and J. W. K. Hong. Towards ONOS-based SDN monitoring using in-band network telemetry. In *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 76–81, Sept 2017.

[14] Netcope. 100G In-band Network Telemetry With Netcope P4. [Online] https://www.netcope.com/getattachment/670aabd2-89f6-4ecf-8620-9b437a256f24/100G-In-band-Network-Telemetry-With-NP4.aspx.

[15] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6. ACM, 2014.

[16] Serkant. Prometheus INT exporter. [Online] https://github.com/serkantul/prometheus_int_exporter.

[17] Blanco Brenden. eXpress Data Path: Getting Linux to 20 Mpps. *Linux Meetup Santa Clara*, July 2016.

[18] Intel. Data Plane Development Kit (DPDK). [Online] https://dpdk.org.

[19] J. Hyun, N. Van Tu, and J. W. K. Hong. INT Management Architecture: ONOS INT Service and XDP. *The 5th P4 Workshop*, June 5, 2018.

[20] InfluxDB: Scalable datastore for metrics, events, and real-time analytics. [Online] `https://github.com/influxdata/influxdb`.

[21] The Prometheus monitoring system and time series database. [Online] `https://github.com/prometheus/prometheus`.

[22] Kernel patches: Bounded loops for eBPF. [Online] `https://www.mail-archive.com/netdev@vger.kernel.org/msg218182.html`.

[23] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, pages 87–95, 2014.

[24] BPF Compiler Collection (BCC). [Online] `https://github.com/iovisor/bcc`.

[25] Grafana: The open platform for beautiful analytics and monitoring. [Online] `https://grafana.com`.

[26] Tcpreplay - Pcap editing and replaying utilities. [Online] `https://tcpreplay.appneta.com`.

# Acknowledgements

First, I would like to express my sincere gratitude to Prof. James Won-Ki Hong for being my supervisor during my Master at POSTECH. His guidance helped me not only to improve my technical knowledge, but also other skills like writing and presentation. His patient and motivation helped to overcome the hard times in my study and research. I also want to thank him for bring me to many places in Korea with the lab trips.

I want to say thank to Prof. Young-Joo Suh and Prof.Hanjun Kim their valuable comments so that I can improve my thesis.

I want to say thank to my labmate in DPNM: Jonghwan for his help and guide in my research, Seyeon, Doyoung, Kyung Chan, Dong Ho, and Vanesco for helping me so much in my life and my study at Postech. I will remember the good moments we have. I want to say thank to June Muk, Ji Bum, Heegon, ChaeHyeon, and Simon, though we do not talk much, but they are always being kind and friendly to me. I also want to say thank to Hieu, who has been my roommate for more than one year and share with me many fun things besides study and research.

Last but not least, I wish to thank my mother Ngo Thi Thua, and my father Nguyen Van Tuan. they have raised me, taught me, and loved me. This thesis is dedicated to them.

# Curriculum Vitae

Name : Nguyen Van, Tu

## Research Interest

Software-Defined Networking (SDN), Network Function Virtualization (NFV), Network Traffic Monitoring and Analysis, Network Programmable Data Plane, Internet of Things (IoT)

## Education

| | |
|---|---|
| 2010 – 2015 | Hanoi University of Science and Technology, Vietnam (B.S.) |
| 2016 – 2018 | Department of Computer Science and Engineering, Pohang University of Science and Technology, Korea (M.S.) |

## Research/Project Experience

2015. 6. – 2016. 7. Embedded system engineer at ICD Limited Liability Company, Hanoi, Vietnam.

## Publications: International Conference

1. J. Hyun, **N. Van Tu**, and J. W. K. Hong, "Towards Knowledge-Defined Networking using In-band Network Telemetry," 2018 Network Operations and Management Symposium (NOMS), Taipei, Taiwan. (Accepted to appear).
2. **N. Van Tu**, J. Hyun and J. W. K. Hong, "Towards ONOS-based SDN monitoring using in-band network telemetry," 2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS), Seoul, 2017, pp. 76-81.
3. **N. Van Tu**, K. Ko and J. W. K. Hong, "Architecture for building hybrid kernel-user space virtual network functions," 2017 13th International Conference on Network and Service Management (CNSM), Tokyo, 2017, pp. 1-6.
4. X. N. Nguyen, **N .V. Tu**, N. N. Pham, M. T. Le, X. N. Tran and V. D. Ngo, "High throughput modified MMSE hardware detector for high-rate spatial modulation systems," 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), Ha Long, 2016, pp. 211-216.

5. **N .V. Tu** , V. T. Thien, S. N. Kim, N. P. Ngoc and T. N. Huu, "A high throughput pipelined hardware architecture for tag sorting in packet fair queuing schedulers," 2015 International Conference on Communications, Management and Telecommunications (ComManTel), DaNang, 2015, pp. 41-45.