A Flexible In-band Network Telemetry Framework for Heterogeneous Private Networks

Gilson Miranda Jr.*†, Jetmir Haxhibeqiri‡, Jeroen Hoebeke‡,
Ingrid Moerman‡, Daniel F. Macedo†, Johann M. Marquez-Barja*
*IDLab - imec, University of Antwerp, Belgium,
‡IDLab - imec, Ghent University, Belgium,
†Universidade Federal de Minas Gerais - Computer Science Department, Brazil
{gilson.miranda,johann.marquez-barja}@uantwerpen.be,
{jetmir.haxhibeqiri,jeroen.hoebeke,ingrid.moerman}@ugent.be,
damacedo@dcc.ufmg.br

***Abstract*—As network management operations increasingly rely on automation and finer control actions, there is a need for precise telemetry systems. In-band Network Telemetry (INT) methods use data packets to carry telemetry and give real-time insights about network performance. Existing solutions often require specialized hardware or offer limited runtime configuration options. This work presents an INT Framework for heterogeneous private networks, targeting industrial and multimedia applications. The framework is designed to be flexible and runtime-reconfigurable, addressing challenges in real-world applications. We provide implementation details of our elements supporting the configurability and the consolidation of raw telemetry into high-level Quality of Service (QoS) metrics. We evaluated the framework in a testbed with wired and wireless devices. The results show the accuracy in monitoring QoS, as well as an analysis of synchronization requirements, showcasing the feasibility of our framework for solutions requiring precise and flexible QoS monitoring.**

***Index Terms*—In-band Network Telemetry, QoS**

I. Introduction

Network management operations are becoming more automated and control actions are being performed in smaller periods, requiring more fine-grained and accurate measurements [1], [2]. Network monitoring is an essential task that affects activities such as network planning, traffic engineering, and troubleshooting. Monitoring can be achieved in different ways, with distinct levels of intrusion, visibility, and accuracy. In-band Network Telemetry (INT) methods use data packets to carry telemetry information, before the data processing and consolidation by collectors [2].

Recent INT methods offer precise insights into network flows, enabling real-time detection of issues such as jitter, latency, packet loss, and uneven load-balancing [3]. Several proposals carry low-level implementations to achieve high throughput [4]. However, such solutions often require specialized hardware and are subject to hardware-imposed limitations. Considering the trend of Network Functions Virtualization (NFV) and the widespread deployment of networked services as containers and virtual machines, there is a potential interest in integrating INT systems into a software stack, fully decoupled from the underlying hardware infrastructure.

Depending on the network and application characteristics, operators need data with distinct levels of granularity and precision, which is beyond the capability of existing techniques [2]. Thus, an INT solution should also be highly flexible and reconfigurable in runtime. For example, during troubleshooting, monitoring frequency can be increased, or intermediate hop monitoring can be toggled on for more granular measurements or off for saving resources, respectively [5].

In this work, we present an INT Framework for heterogeneous private networks, targeting industrial and multimedia applications. Our implementation partially follows the standards in development but deviates in some points as we identify limitations of the existing proposals for our industrial and multimedia use cases. We describe the Application Programming Interface (API) functions implemented for building a highly flexible and runtime-reconfigurable framework. We also address the step of consolidating the raw telemetry data to obtain high-level QoS measurements of *throughput*, *delay*, *jitter*, and *Packet Loss Rate (PLR)*.

This work is organized as follows: Section II reviews related works, focusing on works addressing high-level QoS monitoring. Section III details the framework and its components. Section IV describes the experimental testing setups, and Section V presents the results. Section VI concludes this paper.

II. Background and Related Work

High-level definitions of INT frameworks have been presented in multiple standardization efforts. Song et al. [3] describe a framework for In-situ flow telemetry. Other documents such as RCF9197 and RFC9378 further detail fields and give guidance for the implementation and deployment of In-situ Operations, Administration, and Maintenance (IOAM) systems [6], [7]. RFC8250 describes a set of optional IPv6 headers for End-to-End (E2E) flow monitoring, along with descriptions of fields, limits, data encoding, and metric calculations [8]. These documents provide relevant insights for the development and deployment of INT systems. However, as we progressed with the implementation of the framework, we identified potential optimization points and components that needed further clarification. In Section III we address these points in detail.

Another relevant standardization document was released by P4 [9]. The document provides a series of data plane specifications for INT. The P4 specification gives more detailed low-level specifications and is also an important source of insights for the effective implementation of INT frameworks. Similar to the previous specifications, when a transit node is unable to provide the data specified by a bit, it must either fill the corresponding data field with ones or not add any data at all.

Apart from standardization documents, other works in the literature propose INT implementations tackling issues such as overhead and measurement accuracy. Common approaches are based on selectively or probabilistically deciding whether to add telemetry items to the headers or not [5], [10], [11]. LINT [5] aims at reducing the data plane overhead of INT by selectively reporting data items based on an estimate of accuracy loss. Thus, intermediate nodes may or may not add a certain data item to the packet. A key issue is how to inform the collector node that only a subset of the data was transmitted, and the solution to this is left for future exploration. In our solution, we address this issue along with the problem of inserting blank fields as part of INT data.

PINT [10] follows a different approach and bounds the amount of information added to each packet. The requested telemetry data is probabilistically encoded into several different packets. Instead of having a single packet carrying all the telemetry data, the data are spread over many packets from the flow. PINT targets use cases of congestion control, path tracing, and latency measurements, but for our applications, we require specific measurements of throughput, delay, jitter, and PLR, which in our understanding, cannot be directly obtained with the metadata collected by this method.

Runtime programmability can also be helpful to reduce overhead in addition to giving more flexibility to INT methods. Sel-INT [11] is a runtime-programmable selective INT system. The INT header of Sel-INT has a MapInfo field (equivalent to the trace type of other frameworks) to specify the metadata to be collected at each hop. The MapInfo is 1-byte wide, from which 6 bits are already allocated for metrics such as input/output ports, device ID, hop latency, among others, and a special value of 0xff is reserved to indicate an invalid MapInfo, leaving a single bit available for specification of other metrics. With these fields, we are unable to specify the byte and packet counters necessary to obtain the throughput and PLR metrics that we need for our applications.

A previous version of our framework inserts INT headers between layers 2 and 3, following the header formats that were under specification for IOAM [12]. The focus of the work was on demonstrating the use of INT for monitoring wireless networks and collecting data such as channel, RSSI, and MCS. The work also showed a proof-of-concept of network management using INT feedback. Our current framework aims to combine the strengths of the existing approaches to obtain a highly configurable INT system to measure the key QoS metrics of throughput, delay, jitter, and PLR. We define a per-hop bitmap field that allows the node to indicate which fields it added. As a consequence, this also allows a node to dynamically add a telemetry field, for example, according to a given probability, a threshold, or other conditions. We also define an API to allow runtime reconfiguration of the system to adjust the data collection rate, enable/disable hops, specify how to process reports, and other useful functions.

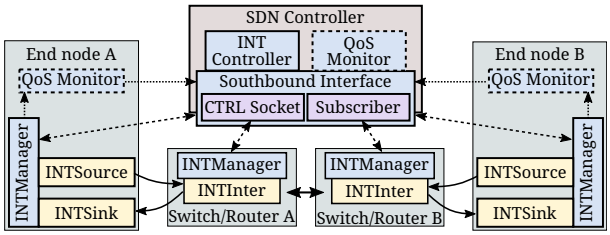

Fig. 1: INT Framework architecture overview

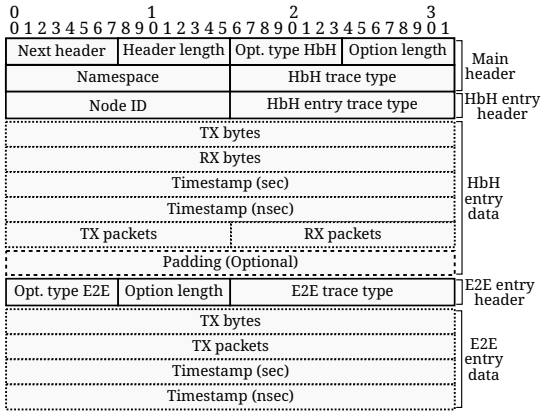

Fig. 2: INT header format as IPv6 extensions

III. INT Framework

Figure 1 illustrates an overview of the framework architecture. The framework can operate without a centralized controller with limited functionality, thus in this paper, we focus on the use case with central control. The framework has six elements: 1) INTManager; 2) INTSource; 3) INTInter; 4) INTSink; 5) INT Controller; and 6) QoS Monitor. Next, we discuss the header format before describing each element.

A. Header Format

Figure 2 shows the header format, along with the necessary E2E and Hop-by-Hop (HbH) fields implemented in this work to collect QoS statistics. The fields of the *Main header* and *E2E entry header* are mandatory. The *HbH entry header* and *entry data* are present when intermediate hops are enabled.

Next header and *Header length* are IPv6 extension header fields specifying the next header in the packet and the extension length. We use 0 for *Next header* (IPv6 Hop-by-Hop Option) [13]. The *Header length* contains the length of the extension header in 8-byte units, excluding the first 8 bytes. The *Option type Hop-by-Hop* with value 0x31 (IOAM Hop-by-Hop Option) [13] denotes the start of the INT headers. The *Option length* represents the size of the HbH data in bytes.

The *Namespace* in our framework has a different semantic compared to RFC9197 [6]. It works as a unique identifier for INT rules, serving as an index for the control structure in

the *INTManager*. The INT Controller uses the *Namespace* for modifying data collection/reporting or removing a rule. If the *Namespace* in a packet does not match an installed rule, the packet is simply forwarded. The *Trace Type* field specifies what information to collect, with a 1 indicating to collect and a 0 to not collect a given metadata. The *Trace Type* is specified as a single 32-bit value but divided into two 16-bit parts: i) *HbH trace type* uses the least significant bits to determine what the *INTInter* elements collect; and the *E2E trace type* with the most significant bits setting what the end nodes collect.

Table I lists the trace type bits implemented. Figure 2 shows the position and size of these fields in the extended headers. Bit 0 adds a 32-bit counter of INT packets generated (for debugging only). Bit 1 adds a 64-bit timestamp from the source node. Bits 2 and 3 add 32-bit counters of transmitted packets and bytes, respectively. Bits 4 to 15 are not allocated yet and can be used for other E2E metadata. For the HbH fields, bit 16 adds two 16-bit counters of transmitted and received packets, respectively. Bit 17 adds a 64-bit timestamp and bit 18 adds 32-bit counters of transmitted and received bytes, respectively. Bits 19 to 31 are not yet allocated. The HbH TX/RX packet fields are 16-bits wide and overflow after 65535 packets. To correctly measure HbH PLR, the rate of packets with INT headers should ideally stay below 65535, which is typically the case in realistic use cases. The processing of the fields and calculation of QoS metrics are detailed in Section III-F.

TABLE I: Trace Type bits

Bit	Name	Mask (Hex)
0	E2E_INTCOUNTER	0x80000000
1	E2E_TIMESTAMP	0x40000000
2	E2E_TXCOUNTER	0x20000000
3	E2E_TX_BYTES	0x10000000
4 to 15	Not allocated	0x0FFF0000
16	HBH_TXRX_COUNTER	0x00008000
17	HBH_TIMESTAMP	0x00004000
18	HBH_TXRX_BYTES	0x00002000
19 to 31	Not allocated	0x00001FFF

The *Node ID* is a unique specifier of the network elements. It is used for identifying intermediate network hops during processing and detecting path changes. This ID is only necessary for intermediate hops, thus, despite being limited to $2^{16} - 1$ bits, it can be set to 0 on all end nodes. The *HbH entry trace type* allows intermediate nodes to add only the information they support to the HbH entry data, instead of populating the unsupported data fields with blank values. If a node can collect all the requested HbH metadata, then the *HbH entry trace type* is equal to the *HbH trace type*. When decoding the data, the INT Sink uses this field to know exactly which fields were added by each node. As a side benefit, intermediate nodes may decide to add only a subset of HbH metadata based on, for example, a probability, or a variation threshold. An optional *Padding* composed of a Header with option type 0x01 (PadN) [13], and a 2-bytes fixed length field is added whenever necessary (depending on trace type) to keep the extended header 8-bytes aligned.

The *E2E entry header* comprises the *Option Type E2E*, *Option length*, and *E2E trace type*. The *Option Type E2E* uses the option type value of 0x11 (IOAM E2E Option-Type) [13]. The *Option length* specifies the byte-length of the *E2E entry data* and *E2E trace type*. The *E2E trace type* field specifies which metadata the end nodes must collect. The E2E entry data carries telemetry collected by the source node. The destination node does not append telemetry but processes the received data to generate a telemetry report.

B. INTManager

The INTManager provides the API for the configuration of the Source/Sink/Inter elements, as well as the control structure with monitoring rule specification and temporary telemetry storage. This is the key element to support runtime programmability of the framework. The API is based on ZeroMQ[1] messaging. Configuration of monitoring rules is done through request/reply sockets, thus every configuration sent by the control plane gets an immediate confirmation. Telemetry reports are transmitted using publish/subscribe sockets. The subscriber runs in server mode at the controller, and the INTManager runs a publisher client.

1) INT Request: To instruct the framework to monitor a flow, the controller sends an INT Rule specification with the format shown in Listing 1, encoded in JavaScript Object Notation (JSON). The message specifies an *int_request*, with the unique Namespace identifier (*ns_id*). The Application identifier (*app_id*) is used to correlate multiple monitored flows of a single application. For example, to monitor both the uplink and downlink flows of a video chat application, we need two rules with distinct *ns_id*s. By attributing the same *app_id* we know that the reports of both *ns_id*s correspond to the video chat application.

Listing 1: INT Request message format

```
"int_request": {
  "ns_id": <number, Namespace ID>,
  "app_id": <number, Application ID>,
  "src_ip": <string, source IPv6 address>,
  "dst_ip": <string, destination IPv6 address>,
  "src_port": <number, source port, 0 for any>,
  "dst_port": <number, dest. port, 0 for any>,
  "protocol": <string, TCP/UDP/ICMP>,
  "trace_type": <string, trace type in Hex.>,
  "mode": <string, count/frequency/probability,
  "mode_setting": <number, value for mode> }
```

The next fields specify the 5-tuple of flow information: IPv6 *Source* and *Destination* addresses, *Protocol* (TCP, UDP, ICMP), *Source* and *Destination* ports. A port value of 0 matches any port. *Trace type* defines the data to be collected (detailed in Section III-A). The mode defines the sampling method (i.e. when the INTSource will generate a packet with INT headers): i) *Packet count* generate an INT-enabled packet every N packets; ii) *Frequency* generates an INT-enabled packet every N milliseconds; iii) *Probability* generates with a probability of $N\%$. The value of N for each mode is specified by the *mode_setting* parameter.

[1]https://zeromq.org/

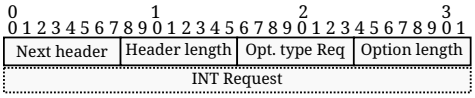

Fig. 3: INT Request header format

After rule installation, the INTSource sends the INT Request in-band to the destination node, embedded in the first data packet of the flow. Figure 3 shows the header format for such a packet. The JSON message is encoded in the INT Request field using Concise Binary Object Representation (CBOR) [14]. We attributed the *Option type* 0x32 to this header. If the complete route is already known by the controller, the rule can be directly installed on all nodes on the path.

2) Namespace Control Structure: The INTManager keeps an internal structure for each monitored flow, described in Listing 2. In addition to the *int_request* fields, the structure contains the packet and byte counters, which are incremented every time a packet of the flow is detected, and other auxiliary fields. The *request_sent* field indicates whether the INT request was already sent in-band to the destination or not. Only after sending this request the actual telemetry data can start being collected. The *last_request_tx* stores the timestamp of the last transmission of the in-band INT Request. The request is re-sent every pre-defined interval (60 seconds by default) to enable the monitoring in new nodes if the route changes. It is also a recovery mechanism in case the first in-band request is lost or if an intermediate node is reset.

Listing 2: Namespace control structure

```
1 struct INTNSCtrl {
2   /* int_request fields (omitted) */
3   /* counters tx/rx bytes/packets (omitted) */
4   bool request_sent;
5   Timestamp last_request_tx;
6   bool transmit_report;
7   bool piggyback_report;
8   bool extract_report;
9   bool hop_enabled;
10  Timestamp last_int_tx; }
```

When a packet with INT metadata reaches the INTSink element, a report is generated and the next variables control what must be done with the JSON-encoded report: i) *transmit_report* field tells whether to publish the report to the controller; ii) *piggyback_report* send the report in-band back to the destination; iii) *extract_report* tells whether an intermediate node should remove a report received in-band (and transmit it to the controller), forwarding the packet without the report to the destination. Next, the *hop_enabled* is used to enable or disable the collection of metadata by an INTInter element, for the specific Namespace. Lastly, the *last_int_tx* marks the transmission time of the last packet, required by the sampling mode based on frequency.

INT reports that must be piggybacked to the source are stored by the INTManager until there is a data packet for the source node. The INTSource requests pending reports and sends them in-band in a packet with headers similar to the shown in Figure 3, but with Option type 0x33. Further details about the reports are presented later in this section. The items of lines 6 to 9 of Listing 2 can be modified through API calls to the INTManager (also *mode* and *mode_setting*). This allows the INT system to be runtime-adjustable, e.g., toggling data collection at certain hops, changing the sampling mode/rate, or specifying which nodes publish telemetry reports.

C. INTSource

The INTSource determines which packets will carry telemetry based on the mode and mode setting. It adds the first extension headers and collects the first E2E metadata at the source. For each packet being transmitted, the INTSource gets the 5-tuple of the flow and queries the INTManager. If a monitoring rule is defined for the flow, the INTManager returns the Namespace control structure for further packet processing and the addition of metadata. If any INT reports or requests are pending to be sent to the destination, they are transmitted instead, with higher priority than telemetry metadata. Before adding the INT headers to any packet, the INTSource verifies if the resulting packet will not exceed the MTU. If this is the case, the extended headers are not added.

D. INTInter

Intermediate nodes (routers, switches, WiFi Access Points (APs)) use the INTInter element to add HbH telemetry data to packets. If a packet carries an INT request, the request structure is processed and the Namespace control structure is created by the INTManager. The metadata collection by the INTInter element can be enabled or disabled by the controller through the INTManager, for each Namespace. For example, if the E2E performance of a flow is satisfactory, HbH metadata collection can be disabled to reduce data and processing overhead. If necessary, the intermediate hops can be selectively enabled to discover which hop is degrading the performance.

When a packet with the INT header is received the element verifies if there are HbH bits configured in the trace type and if the addition of the metadata will not exceed the MTU. If a given metadata is specified by the trace type but cannot be obtained by the node, the respective bit of the *HbH entry trace type* is disabled. The INTInter stacks its metadata just after the main header (Figure 2), stacking over data from the previous hop. The INTSink uses this ordering scheme and the *Node ID* when it generates the INT report in a way that the QoS Monitor can interpret the data and calculate the QoS metrics for each pair of nodes between the source and destination.

E. INTSink

The INTSink decodes and processes the metadata, generating an INT report. For each hop, it extracts the HbH entry data to know exactly which fields were added by the INTInter element. A report in JSON format is created and depending on how the Namespace is configured, the report is either stored at the INTManager or published to the controller. All the extension headers are removed and the packet is further processed at the higher layers. In the following sections, the report generated by the INTSink is referred to as *raw report*.

F. QoS Monitor

The raw INT reports contain values of packet/byte counters, timestamps, and node ordering information. To obtain meaningful QoS metrics we need further aggregation and processing of reports by the QoS Monitor. We implement it as a separate module that can be deployed either at the controller or at the end nodes. Raw reports are generated for every INT-enabled packet, thus, depending on sampling rate and flow characteristics, this can generate a high overhead of reports published to the controller. Instead, consolidated QoS reports published once per second are sufficient for most applications.

The QoS Monitor can be configured to calculate the QoS metrics of throughput, delay, jitter, and PLR in multiple timescales (e.g., 1 second, 10 seconds, 1 minute). For that, the QoS Monitor always keeps a record of the most recent raw reports of each Namespace (1000 samples used around 1 MB of RAM in our measurements). By default, it calculates the mean values for the last one second. From the four QoS metrics, only the delay can be calculated with a single raw report, from the difference of the timestamps from each hop. The other metrics require at least two raw reports. If the sampling rate is insufficient, the values are averaged over the period between the last two reports.

Algorithm 1 describes the throughput calculation. First, we get the last entry in the table of raw reports, the timestamp of the last entry, and the RX Bytes of the last entry (lines 1, 2, and 3). Line 4 gets the start timestamp, based on the calculation period demanded, and line 5 gets the first valid entry near the $baseTS$. Lines 6 and 7 get the RX Bytes and timestamp of the start entry. With these values, we can calculate the transmitted bytes ($endBytes - startBytes$) during a period ($endTS - startTS$). The condition in line tests if there was an overflow between the first and last entry and adjusts the counter based on the maximum possible value of the field. The last two lines take the difference of the byte counter over time and return the bytes per second calculation.

Algorithm 1: Calculate Reception Throughput

1: $lastEntry \leftarrow table[numEntries]$
2: $endTS \leftarrow lastEntry['timestamp']$
3: $lastBytes \leftarrow lastData['rxBytes']$
4: $baseTS \leftarrow endTS - period$
5: $startData \leftarrow get\_first\_entry(baseTS)$
6: $startBytes \leftarrow startData['rxBytes']$
7: $startTS \leftarrow startData['timestamp']$
8: **if** $endBytes < startBytes$ **then**
9: $\quad endBytes \leftarrow endBytes + 4294967296$
10: **end if**
11: $difference \leftarrow endBytes - startBytes$
12: **return** $difference/(endTS - startTS)$

The PLR calculation follows the same approach from Algorithm 1, but using the packet counters, and the difference between transmitted and received packets over a period. The overflow adjustment done in line 9 takes into account whether the packet counters are from HbH or E2E measurements. If the counter is from HbH, the value added is 65536 (the maximum for the 16-bit counter). Lastly, the jitter is calculated through the difference of sequential samples of delay, also requiring at least two raw INT reports. The accuracy of timing measurements (i.e. delay and jitter) depends on clock synchronization between the nodes. In Section V we analyze the measurement accuracy using different time synchronization methods.

IV. Experimental Setup

This section details the experimental setup in terms of topology and hardware. Figure 4 gives an overview of the topology, consisting of 5 hops, with one source, one destination, and four switches. For most of the experiments, we used a fully wired setup, with 2 wired nodes and 4 switches. The switches are generic industrial mini-pcs with Intel Celeron J4125 @ 2.0GHz, 8 GB of RAM, and 4x Intel I225-V Ethernet interfaces. The Source and Destination nodes were Intel NUCs model NUC11TNH. The nodes were synchronized via Precision Time Protocol (PTP) with the controller. For another set of experiments, we configured SW4 as a Wi-Fi AP, and the destination node was a Wi-Fi Client. We further specify the test setup in each subsection of Section V.

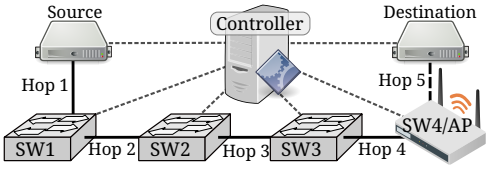

Fig. 4: Topology used for experimental evaluation

A. Software tools

The INTManager, INTSource, INTInter, and INTSink were created as elements of Click Router [15]. The INT Controller and the QoS Monitor were implemented in Python. All experiments used trace type 0x7000E000, enabling E2E bits 1, 2, 3 and HbH bits 16, 17, and 18. We used three tools for traffic generation: Multi-Generator (MGEN) Network Test Tool[2], Iperf3[3], and ping. We used MGEN with two types of UDP traffic: i) *constant*, sending 10000 Packets Per Second (pps), each with 500 bytes of payload, resulting in a throughput of 44.96 Mbps; ii) *bursty*, interleaving bursts of packet transmissions with silent periods, where it generates a burst of 10000 pps of 500 bytes for 2 seconds. Each burst occurs in random intervals with a mean duration of 5 seconds. We used Iperf and ping to evaluate the performance impact, and accuracy with NTP synchronization, respectively.

B. Test Cases

To assess monitoring accuracy, we devised test cases focusing on different QoS metrics. Using Traffic Control (TC) Network Emulator (NETEM) (tc-netem[4]), we applied performance impairments on different hops and compared the impairment values with measurements from the INT framework.

[2]www.nrl.navy.mil/itd/ncs/products/mgen
[3]https://iperf.fr/
[4]https://man7.org/linux/man-pages/man8/tc-netem.8.html

For this comparison, we account for the composition mode of metrics along the path. The metrics of interest (i.e., throughput, delay, jitter, and PLR) are composed in distinct ways [16]. Considering a path $p = (i, j, k, ..., l, m)$ the compositions are as follows: i) Throughput is concave, described by Equation 1, being limited by the minimum throughput among all pairs in the path between i and m; ii) Delay and Jitter are additive. The values accumulate along the path, as described in Equation 2; iii) The composition for PLR probability is multiplicative, as described by Equation 3.

$$T(p) = min[t_{(i,j)}, t_{(j,k)}, ..., t_{(l,m)}] \quad (1)$$

$$D(p) = d_{(i,j)} + d_{(j,k)} + ... + d_{(l,m)} \quad (2)$$

$$L(p) = 1 - ((1 - l_{(i,j)}) \times (1 - l_{(j,k)}) \times ... \times (1 - l_{(l,m)})) \quad (3)$$

We use these equations to verify whether the measurements at each hop are in line with the expected values, according to the impairment set. Table II describes the four test cases and the impairments set on each hop. Test case 1 focuses on throughput, starting with 100 Mbps in the first hop, and gradually decreasing to 10 Mbps in the fourth hop. Test case 2 adds delays at each hop, but no throughput limitations. Test case 3 adds jitter to the delay values from test case 2. Test case 4 inserts different PLR probabilities at each hop.

TABLE II: Test cases

Test Case	Metric	Hop				
		1	2	3	4	5
1	Throughput (Mbps)	100	40	20	10	100
2	Delay (ms)	10	5	5	10	20
3	Delay ± Jitter (ms)	10±2	5±1	5±1	5±2	20±4
4	PLR (%)	5	2	5	1	2

V. Results

A. Measurement accuracy

We evaluate measurement accuracy by calculating the Mean Absolute Error (MAE) between measurements reported by the framework and the impairments applied with tc-netem. The jitter and PLR measurements are expected to be slightly imprecise due to how tc-netem operates. The loss rate set in tc-netem defines the probability of dropping each packet before transmission, thus, the real loss is variable during the experiment. For jitter, the specified value is the upper bound of variation to be added or subtracted to the delay, for each packet to be transmitted. This also causes variations between the delay value configured and the real network delay.

Table III shows the results for throughput measurements, using the three sampling strategies (packet count, frequency, and probability). Each row shows the hop, the expected throughput, and the measurements with each strategy. The last row shows the MAE over the 120 QoS samples collected, with a mean error of under 0.50 Mbps for all modes. Figure 5 shows the measurements for the other metrics during the same set of experiments. As tc-netem delays the packet transmissions to achieve the set throughput, we also observe different delays, jitter, and packet losses caused by queue overflows in bottleneck links. Figure 5 shows the mean values of throughput, delay, and PLR, along with the distribution of jitters during the experiment with packet count strategy.

TABLE III: Throughput Measurements

Hop	Expected (Mbps)	Measured (Mbps)		
		Count	Frequency	Probability
1	44.96	44.41	43.81	43.81
2	40	39.56	39.10	39.10
3	20	19.85	19.70	19.70
4	10	10.00	9.99	9.99
5	10	9.76	9.74	9.74
E2E	10	9.76	9.74	9.74
MAE		0.2830	0.4916	0.4950

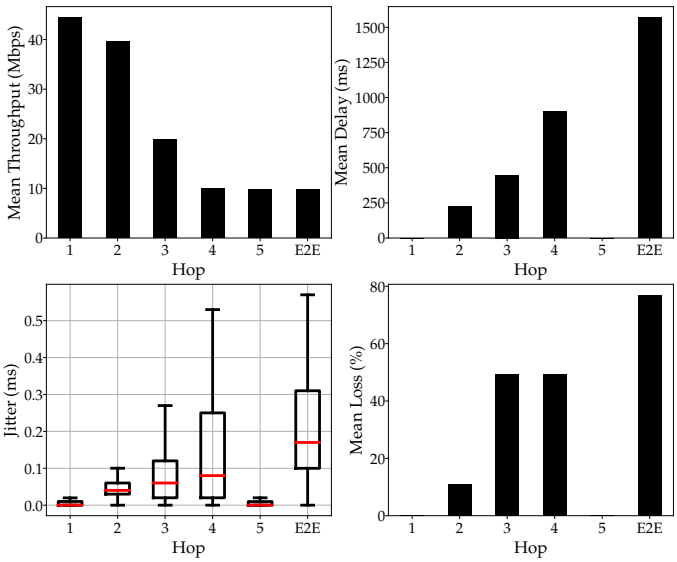

Fig. 5: Measurements in restricted throughput scenario

At Hop 1 (unlimited throughput) we measured between 43.81 and 44.41 Mbps, slightly below the expected 44.96 Mbps. Hop 2 limits throughput to 40 Mbps, and measurements range from 39.10 to 39.56 Mbps. The mean delay is 221 ms with a 10 % PLR. Hop 3 further restricts throughput to 20 Mbps, measured between 19.70 and 19.85 Mbps, with a delay of 448 ms and 49 % PLR. Hop 4 limits throughput to 10 Mbps, with measurements between 9.74 and 9.76 Mbps. The mean delay on Hop 4 is 898 ms, and PLR is 49 %. Jitter distribution (Figure 5) is consistently below 1 ms in all hops. E2E measurements reflect the composition of intermediate hops: throughput is limited to 10 Mbps, E2E delay is the sum of intermediate delays (1568 ms), and the PLR is 76 %, aligned with the expected composition from Equation 3.

Tables IVa and IVb show the results for delay without jitter, and delay with jitter, respectively, and Table IVc shows the results for PLR. Each table shows the value *configured* with tc-netem at each hop, and the INT measurements for constant and bursty traffic obtained using the *packet count* strategy. Each row shows the result for a given hop. The last two rows

show the E2E measurements expected (according to Equations 1, 2, and 3) and measured by the framework, and the MAE over the 120 data points collected.

TABLE IV: Delay, Jitter and Loss measurement accuracy

(a) Delay Measurements

Hop	Configured (ms)	Measured (ms)	
		Constant	Bursty
1	10	10.02	10.03
2	5	5.03	5.02
3	5	5.02	5.02
4	10	10.02	10.02
5	20	20.01	20.01
E2E	50	50.09	50.09
MAE		0.0310	0.0315

(b) Delay and Jitter Measurements

Hop	Configured (ms)		Measured (ms)			
			Constant		Bursty	
	Delay	Jitter	Delay	Jitter	Delay	Jitter
1	10	2	10.23	1.16	10.14	1.29
2	5	1	5.01	0.70	5.10	0.61
3	5	1	5.07	0.59	5.11	0.76
4	10	2	10.22	1.19	10.05	1.26
5	20	4	20.50	2.80	20.55	2.91
E2E	50	5	51.04	3.43	50.94	3.39
MAE	-	-	1.3026	1.2815	1.2683	1.2511

(c) Packet Loss Ratio Measurements

Hop	Configured (%)	Measured (%)	
		Constant	Bursty
1	5	5.01	4.99
2	2	2.00	2.03
3	5	5.00	4.96
4	1	1.00	1.02
5	2	1.99	2.02
E2E	14.2	14.19	14.20
MAE		0.1628	0.1696

In Table IVa, the MAE of delay measurements was under 0.03 ms for both traffic types. When jitter is present (Table IVb), the MAE increases slightly due to the sampling rate of INT and how tc-netem inserts the jitter (detailed earlier in this section). Still, the framework allows us to identify which hops contribute the most to jitter. For PLR (Table IVc), INT can precisely identify the HbH and E2E PLR with MAE under 0.17 %. As we use packet counters to detect packet losses, the system is robust to losses of packets carrying INT data, as long as at least some of those packets traverse the E2E link.

B. Delay measurements with NTP synchronization

The previous experiments were carried out on a wired testbed with nodes synchronized with PTP. However, in many cases, it is infeasible to use PTP (e.g., with Wi-Fi 6 and older standards). We conducted a set of tests using the SW4 as a Wi-Fi AP[5] and replaced the destination node with a laptop[6]. We generated pings with intervals of 0.01 s from the Source to the Destination and monitored both directions with INT. We set up an NTP server at the controller, to which we synchronized the nodes with varying intervals (60, 30, 10, 5, and 1 second).

We analyzed the accuracy between the Round-Trip Time (RTT) measurements returned by ping, and the RTT measured by INT (sum of one-way delays). Table V shows the MAE according to synchronization frequency. Without synchronization, the MAE was 0.216 ms. However, the analysis of one-way delays showed values such as -36 seconds, i.e., the clock of the Source node was 36 seconds in the future, compared to the clock of the Destination. Despite giving accurate RTT values, the one-way delays measured were not reliable. With NTP synchronization, the MAE was under 650 μs.

TABLE V: RTT measurement error between reports by ping and INT

Frequency	None	60s	30s	10s	5s	1s
MAE (ms)	0.216	0.056	0.052	0.052	0.051	0.065

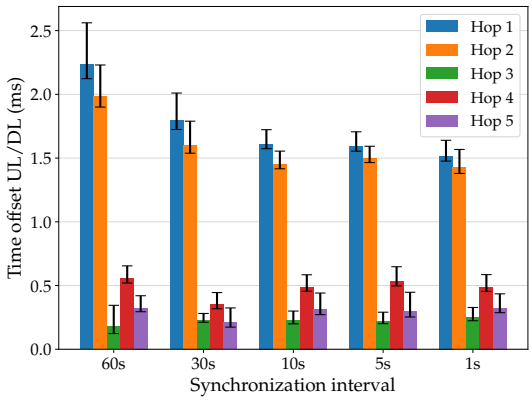

Fig. 6: Uplink/Downlink offset with NTP synchronization

Figure 6 shows the absolute offset between the measurements in uplink and downlink at the same hop, at the same time. Each hop was tagged according to the topology of Figure 4. We observed higher offsets in hops 1 and 2, which were reduced by increasing the synchronization frequency. The offset was under 0.7 ms for the other hops in all cases. The higher offsets of hops 1 and 2 might have been caused by link asymmetry, despite those hops being wired. Nonetheless, delay measurement errors of 1.5 ms might still be acceptable in use cases with high network load, when the experienced delays can be in the order of tens of milliseconds. In those cases, the framework can still help to identify bottlenecks.

C. Performance impact

INT monitoring comes with a performance cost from additional packet processing. We analyzed the performance degradation versus sampling rate using a simplified topology with SW2 as an end node. We generated iperf traffic from the Source, passing through SW1, until the server on SW2. The maximum link capacity is 2.5 Gbps, but as we used Click without optimizations such as eXpress Data Path (XDP) [17],

[5]Wi-Fi 5 mode with a Sparklan WNFT-238AX(BT) card, Ubuntu Server 22.04, Linux Kernel 6.4, Hostapd v2.11-devel

[6]Dell Inspiron 13-7000, with standard Kubuntu 23.10

the base performance that we achieved was around 1.33 Gbps. We executed 30 iperf runs of 30 seconds for each test scenario.

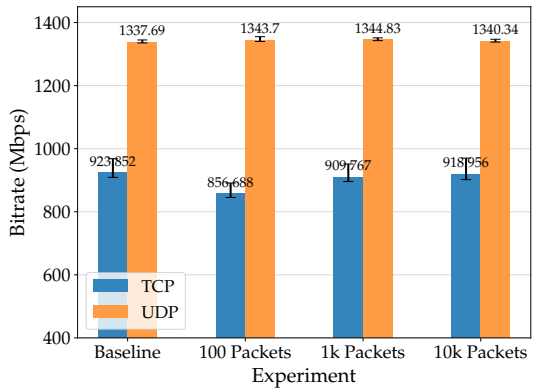

Fig. 7: Performance impact of monitoring frequency

Figure 7 shows the throughput for each sampling rate. There is a performance drop of 7.27 % for the TCP flow when adding INT once every 100 packets (1/100). Reducing this rate to 1/1000 packets improves the throughput by 6.19 %, and reducing it to 1/10000 improves the throughput by 7.26 %. No significant changes were observed for UDP flows. These results show how the ability to update the sampling rate in runtime can help to reduce overhead. If E2E performance degradation is detected, the sampling rate can be adjusted to help identify the bottleneck. For high-throughput flows or if the sampling rate is misconfigured during setup, the sampling rate can later be updated to a more suitable value.

VI. Conclusion

This paper presents an INT framework focused on flexible configuration for efficient data collection in heterogeneous networks. We provide a detailed description of the framework, covering implementation decisions, such as messaging library, internal control structures, configuration structures, and algorithms for extracting high-level QoS metrics from raw reports. We conducted extensive measurements in a real testbed with wired and Wi-Fi topologies, and different time synchronization methods. The fully software-based components are suitable for solutions based on virtualized elements such as containers and VMs that require precise and flexible QoS monitoring but do not have access to advanced programmable hardware features.

Acknowledgment

This research is partially funded by the imec ICON project VELOCe - VErifiable, LOw-latency audio Communication (Agentschap Innoveren en Ondernemen project nr. HBC.2021.0657). This research is also supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and São Paulo Research Foundation (FAPESP) with Brazilian Internet Steering Committee (CGI.br), grants 2018/23097-3 and 2020/05182-3.

References

[1] M. H. Behringer, M. Pritikin, S. Bjarnason, A. Clemm, B. E. Carpenter, S. Jiang, and L. Ciavaglia, "Autonomic Networking: Definitions and Design Goals," RFC 7575, Jun. 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7575

[2] H. Song, F. Qin, P. Martinez-Julia, L. Ciavaglia, and A. Wang, "Network Telemetry Framework," Internet Engineering Task Force, Internet-Draft draft-ietf-opsawg-ntf-07, feb 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-opsawg-ntf-07

[3] H. Song, F. Qin, H. Chen, J. Jin, and J. Shin, "A Framework for In-situ Flow Information Telemetry," Internet Engineering Task Force, Internet-Draft draft-song-opsawg-ifit-framework-17, aug 2022. [Online]. Available: https://www.ietf.org/archive/id/draft-song-opsawg-ifit-framework-21.html

[4] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li, "In-band Network Telemetry: A Survey," *Computer Networks*, vol. 186, no. August 2020, 2021. [Online]. Available: http://doi.org/10.1016/j.comnet.2020.107763

[5] S. R. Chowdhury, R. Boutaba, and J. Francois, "LINT: Accuracy-adaptive and Lightweight In-band Network Telemetry," *Proceedings of the IM 2021 - 2021 IFIP/IEEE International Symposium on Integrated Network Management*, no. Im, pp. 349–357, 2021. [Online]. Available: https://inria.hal.science/hal-03525026

[6] F. Brockners, S. Bhandari, and T. Mizrahi, "Data Fields for In Situ Operations, Administration, and Maintenance (IOAM)," RFC 9197, pp. 1–40, may 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9197

[7] F. Brockners, S. Bhandari, D. Bernier, and T. Mizrahi, "In Situ Operations, Administration, and Maintenance (IOAM) Deployment," RFC 9378, Apr. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9378

[8] N. Elkins, R. Hamilton, and M. Ackermann, "IPv6 Performance and Diagnostic Metrics (PDM) Destination Option," RFC 8250, sep 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8250.txt

[9] The P4.org Working Group, "In-band Network Telemetry (INT) Dataplane Specification," *The P4.org Applications Working Group*, pp. 1–42, 2020. [Online]. Available: https://p4.org/p4-spec/docs/INT_v2_1.pdf

[10] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "Pint: Probabilistic in-band network telemetry," ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 662–680. [Online]. Available: https://doi.org/10.1145/3387514.3405894

[11] S. Tang, D. Li, B. Niu, J. Peng, and Z. Zhu, "Sel-INT: A Runtime-Programmable Selective In-Band Network Telemetry System," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 708–721, 2020. [Online]. Available: http://doi.org/10.1109/TNSM.2019.2953327

[12] J. Haxhibeqiri, P. H. Isolani, J. M. Marquez-Barja, I. Moerman, and J. Hoebeke, "In-band network monitoring technique to support sdn-based wireless networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 627–641, 2021. [Online]. Available: http://doi.org/10.1109/TNSM.2020.3044415

[13] "Internet Protocol Version 6 (IPv6) Parameters," https://www.iana.org/assignments/ipv6-parameters/ipv6-parameters.xml, accessed: 2023-12-12.

[14] C. Bormann and P. E. Hoffman, "Concise Binary Object Representation (CBOR)," RFC 8949, Dec. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8949

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, p. 263–297, aug 2000. [Online]. Available: https://doi.org/10.1145/354871.354874

[16] Zheng Wang and J. Crowcroft, "Quality-of-service routing for supporting multimedia applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1228–1234, 1996. [Online]. Available: http://doi.org/10.1109/49.536364

[17] C. S. Barbette Tom and M. Laurent, "Fast userspace packet processing," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2015, pp. 5–16. [Online]. Available: http://doi.org/10.1109/ANCS.2015.7110116