A Practical Implementation of In-Band Network Telemetry in Open vSwitch

Anton Gulenko, Marcel Wallschläger, Odej Kao
Complex and Distributed IT-Systems, Technical University Berlin
{firstname}.{lastname}@tu-berlin.de

***Abstract*—Virtualized network infrastructures of cloud platforms are becoming increasingly complex and hard to manage and monitor for human administrators. Traditional network monitoring approaches like streaming telemetry or active probes are of limited applicability in highly virtualized environments and introduce unwanted overhead. The recent concept of In-Band Network Telemetry (INT) allows detailed observation of physical and virtual networking components exactly when real user traffic traverses them. This approach eliminates the need for artificial probing packets, allows data collection on the level of individual virtual bridge ports, and supports advanced applications like multipath reconstruction, detection of dead hops, and localization of latency bottlenecks. Our prototypical extension of the de-facto standard virtual switch Open vSwitch shows the low resource overhead of INT in an experimental evaluation.**

***Keywords*—Cloud Computing, Network Monitoring, In-Band Network Telemetry**

I. Introduction

The SDN (Software Defined Networking) methodology allows dynamic and flexible runtime changes to network infrastructures. These capabilities are heavily used to implement the virtualized network fabric that connects virtual machines (VMs) in cloud infrastructures. One important benefit of SDN enabled networks is a logically centralized controller, which enables scaling up the network infrastructure to large extents.

Virtual switches play a special role in cloud networks, as they organize the network tunnels and connections between virtual machines and the outside world. The most prominent virtual switch implementation is Open vSwitch (OVS), as it is the default network layer used by OpenStack clouds, the industry standard open source cloud stack.

Modern cloud applications have a very high demand for powerful reliability mechanisms and in-depth monitoring. Virtualized services running inside a cloud deployed on commodity hardware frequently experience both crashes and performance anomalies due to software bugs or hardware malfunctions. Such anomalies can be detected by analyzing metrics sampled from all layers of the infrastructure. Basic performance related metrics can be obtained directly from the operating system running the services, or from the virtualization layer that provides virtual machine functionality [1]. One additional important source of data is the network.

Traditionally, there are three approaches to network monitoring: streaming telemetry, polling of statistics from network devices, or active probing packets. In the former two cases, the obtained data is either a sampling of the processed network traffic, or statistics about the state of the network device like processing queue lengths, number of processed packets, or the utilization of ingress or egress ports. The active probing approach involves sending special monitoring packets to the services or network elements and measure data like response time. This technique has multiple drawbacks: it introduces additional network traffic overhead, affects the state of the observed device or service, and is highly service specific, making it less portable between services and necessitating new monitoring techniques for every service or network device. In-band Network Telemetry (INT) [2], [3] is a novel network monitoring technique that does not include the negative aspects of traditional monitoring systems. With INT, monitoring data is attached to user traffic packets by every network element as they traverse it. The data is attached as metadata, and therefore remains invisible to the user applications. At the network egress points, where the packets leave the monitoring network domain, the metadata is stripped from the packet and forwarded to a monitoring host.

The INT approach introduces three main benefits to network monitoring:

1) No artificial probing packets are required
2) The network state is observed at the exact point in time the real user traffic passes through it
3) Very fine grained control over the amount of monitoring traffic and over the metered connections is possible

As INT is not yet a widely used industry standard, there is no open source software switch that natively implements the INT technology. This paper presents a prototypical implementation of INT in the software switch Open vSwitch and evaluates the prototype regarding the expressiveness of the collected data and performance overhead.

The rest of this paper is organized as follows. Section II describes the INT methodology and the format of INT metadata. Section III introduces details of our prototypical implementation of INT in Open vSwitch, while section IV presents an evaluation of the performance overhead added by our extension. Finally, section V concludes and discusses possible improvements and future efforts.

II. In-Band Network Telemetry

In-Band Network Telemetry (INT) is a novel approach first described by the authors of the P4 programming language [3].

The core idea is to attach monitoring information directly to application traffic flowing through the different network devices. The host or device that generates the traffic of interest initiates the data collection by attaching INT commands to the outgoing traffic. These commands are attached in a way transparent to application services, meaning the service layer is in no way disturbed by the process. Every network device handling packets with INT command headers will attach the requested information using additional INT data headers. The final receiver of the packets can unpack the collected data and forward it for further visualization and analysis.

The INT methodology brings many advantages that can be used for SDN network tomography and monitoring:

- Works with non-traditional network elements like SDN switches and software switches
- Transparent for service layer traffic
- Delivers monitoring information directly when the service traffic is processed by the network devices
- Low and controllable network overhead

INT data is collected by inserting an INT request header into a user traffic packet. The format for this header is defined in [3] and shown in Figure 1.

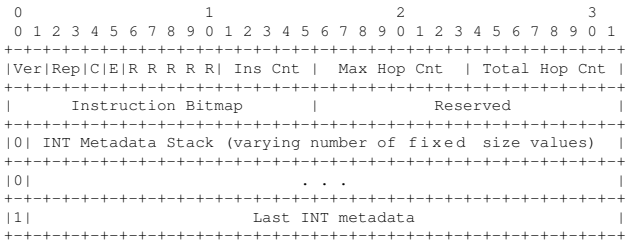

Fig. 1. INT request header and data

The INT request header size is 8 byte. The header contains miscellaneous maintenance fields like the protocol version and counters for the number of INT hops this packet has traversed. After that, the *Instruction Bitmap* denotes the metrics that each INT transit hop is requested to append to the packet. The following metrics can be requested by setting individual bits: Switch ID, Ingress port ID, Hop latency, Queue occupancy, Ingress timestamp, Egress port ID, Queue congestion status, Egress port utilization.

Besides the format of the INT metadata, the transport methodology must be specified and supported by all devices in a network domain. One typical issue when implementing INT transport is that packets have to be grown while they travel through the network. Growing packets by an unpredictable size can lead to fragmentation due to the Maximum Transmission Unit (MTU) of the network, which must be considered when planning and configuring the network. Further, a good transport methodology should be transparent to user level traffic to take full advantage of INT capabilities.

Two main transport protocols have been proposed for INT [3]. The first one is Geneve, a generic extensible tunneling framework. The second one is VXLAN, a common tunneling protocol supported by most virtual and physical switches and standardized in RFC 7348. The authors propose to embed INT metadata in a shim header between the VXLAN header and the encapsulated payload. This allows for transparent and extensible transport of metadata, but the current RFC 7348 assumes that the payload following the VXLAN header is always an Ethernet header. An IETF draft [4] proposes changes to the VXLAN header to allow for multiprotocol encapsulation.

A. Implementation Strategies

As the variety of virtual network technologies has evolved, there is a large number of possible implementation strategies for INT support. This section discusses a selected set of alternatives to the chosen strategy of extending the Open vSwitch code base. The two most important parameters when selecting an implementation approach are feasibility and performance. Feasibility includes the prospect of the implementation being accepted and used in practice, as well as the question, whether the required functionality can actually be achieved. The performance parameter is important since processing of network packets is often a performance critical matter, and the overhead generated by a monitoring solution be be bounded to consider practical use.

The extended Berkeley Packet Filter (eBPF) [5], [6] is a technology for attaching user-defined byte code to certain points in the Linux Kernel that will be executed within kernel mode without context switch overhead. One of the most common use cases for eBPF is to define a short routine that inspects or modifies all packets processed by a certain network interface or Linux bridge. This technology can be leveraged to implement parts of the INT infrastructure outside of Open vSwitch. While showing great potential, this technology alone is not enough to implement the end-to-end INT infrastructure due to the following problems:

1) eBPF code cannot be attached to Open vSwitch bridges and ports, only to Linux bridges and real network interfaces. This greatly limits the scope of the collected INT data. Tunnels between virtual machines are terminated by Open vSwitch, which means that INT command headers inserted by an eBPF program will miss the packet path from the tunnel endpoint until the packet leaves the physical host.
2) eBPF cannot efficiently access internal Open vSwitch data. Interesting data can include queue sizes, congestion and load data, error states, and so on. This problem could be solved through a userspace daemon extension to OVS that copies the OVS internal data into kernel-maps accessible to eBPF programs. However, this indirection would negate the advantage of INT that it reflects the switch state of the exact moment when the packet is processed.

Another approach for implementing an INT-capable virtual switch is the preliminary INT spec published by the authors of P4 [3] as a primary use case of the flexibility of the P4 programming language [7]. P4 is a descriptive language

for network devices like switches or routers. Some of the current physical devices on market support to be programmed by P4. P4 is available including an implementation of the INT specification using VXLAN as the transport protocol for the INT headers. Provided a fully P4-compatible software switch, this reference P4 program could be used to set up an INT-capable infrastructure. Currently P4 programmable software switches are in a early stage and did not find their way into the industry [8]. Further, implementing INT directly inside the kernel module C code would also result in reduced performance overhead.

Finally, a combination of the eBPF and P4 technologies can be considered by compiling P4 code to eBPF byte code and loading the resulting program directly into the relevant places in the kernel. However, due to the restrictions for eBPF programs, like limited looping functionality or enforced runtime guarantees, the existing P4 reference programs cannot be compiled directly.

III. INT-capable Open vSwitch Prototype

An Open vSwitch installation on a Linux server consists of a kernel module and two main user space processes, accompanied by a number of command line tools for administration and configuration. For feature extensions like implementing INT in Open vSwitch, all of these components must be extended. Figure 2 gives a high-level overview over the components of Open vSwitch.

One of the strengths of OVS is that it can be controlled via the OpenFlow protocol, making it a native addition to any SDN-based network. To leverage the OpenFlow capabilities of OVS in our INT extension, a dedicated OpenFlow action was added to all layers of the Open vSwitch architecture. An SDN controller that is aware of the added INT action can automatically insert and configure INT header processing in any flow inside the virtual switch. Since currently no SDN controller is supporting this custom action, the INT actions must be inserted manually, in a way that does not disturb the regular operation of the virtual switch. The action can be inserted with the following command:

```
ovs-ofctl add-flow br1 \
    "priority=100,
    udp,
    nw_dst=192.168.100.100,
    actions=int_transit(300),
    NORMAL"
```

In this example, all UDP packets with a destination IP of `192.168.100.100` will be handled by the INT action, and then forwarded to the regular switching logic. The OpenFlow action name is `int_transit`, and the parameter `300` is the ID of the inserted action, that can be queried through the `Switch ID` flag in an INT request header. These IDs should be managed by the SDN controller in a fully implemented INT environment.

Aside of adding an OpenFlow action, the port functionality of Open vSwitch must be extended as well. OVS receives

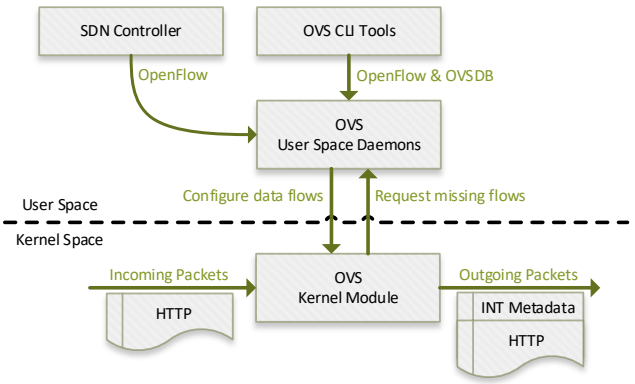

Fig. 2. Open vSwitch basic architecture

packets on an *ingress port*, passes them through a list of associated OpenFlow tables, and finally outputs packets through one or more *egress ports*. In order to implement a more complete set of supported INT metrics, the port functionality must be extended as visualized in Figure 3:

1) At the ingress port, some INT-related metadata is allocated and associated with the packet. An *ingress timestamp* with nanosecond granularity is stored in the reserved memory.
2) If the packet is handled by an INT OpenFlow action, a flag is set within the associated INT metadata.
3) At the egress port, packets with the enabled INT flag are handled specially. The INT request header is parsed, and the requested INT metrics are computed. For this, the ingress timestamp stored in the INT metadata is used, as well as further data which is maintained by the OVS code.

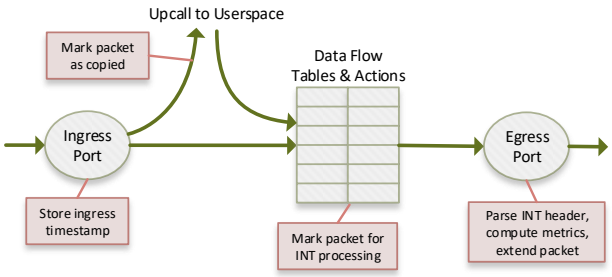

Fig. 3. Added functionality at Open vSwitch ports and OpenFlow actions

The INT metrics supported by the prototype are shown in Table I. Most metrics implement the default metrics suggested by the INT specification. The notable exception is the metric *Packet Copied to Userspace*, which is specific to Open vSwitch. It is a flag that is set to `1`, when the annotated packet has been copied to the user space for further processing, and to `0` otherwise. This happens when the OVS kernel module has no cached information about how it should handle the packet. In this case, the kernel module requests instructions from the user space daemons that have a complete view of all

OpenFlow tables, rules and actions. This additional metric is useful for detecting misconfigurations and performance issues in virtual switches.

TABLE I
SUPPORTED INT COMMANDS

Bit	Metric name	Data Source
0	Switch ID	Set by the user as parameter of the `int_transit` OpenFlow action
1	Ingress port ID	Maintained by OVS
2	Hop Latency	Difference between timestamps taken at ingress and egress ports
3	*Not implemented*	—
4	Ingress Timestamp (s)	Taken at ingress port
5	Egress port ID	Maintained by OVS
6	*Not implemented*	—
7	Egress port tx utilization	Maintained by OVS
8	Ingress Timestamp (μs)	Taken at ingress port
9	Ingress Timestamp (ms)	Taken at ingress port
10	Packet Copied to Userspace	Maintained by OVS
11	Egress Port Packets	Maintained by OVS
12	Egress Port Errors	Maintained by OVS
13	Egress Port Dropped	Maintained by OVS
14	Egress Packets per Second	Maintained by OVS (aggregated)
15	Egress Bytes per Second	Maintained by OVS (aggregated)

For the sake of simplicity of this research prototype, the INT metadata is transported over UDP. More precisely, the monitored data is written directly into the payload of a preallocated UDP packet of sufficient size. This implies that only UDP packets can be handled by the new OpenFlow action, and that user traffic can not transparently be monitored. Since this disables one of the strong aspects of INT compared to other monitoring techniques, an transport implementation over a tunneling protocol like VXLAN or GRE is the next step for making the evaluated prototype usable in a wider range of use cases.

IV. EVALUATION

An evaluated for the presented prototype was conducted on an installation of the modified OVS on a dedicated server with 2GB of memory and 2 CPU cores. The server was configured to forward traffic of a certain IP subnet to a subsequent host. The forwarded traffic was routed through an OpenFlow table in Open vSwitch, which contained the introduced `int_transit` OpenFlow action. In order to enable the UDP transport for the INT transit hop, specially crafted UDP packets were sent to an IP inside the forwarded subnet. The crafted UDP packets contained a valid INT metadata header that requested all INT transit hops to collect the entire set of available metrics Upon parsing these UDP packets, the modified OVS kernel module wrote the requested information directly into the payload of the packet. On the host with the destination IP of the extended packets, a UDP server would receive and extract the collected information, and store it for later evaluation.

During the experiment, the stream of UDP packets was set to different levels of throughput between 1 and 100 MByte/s.

The evaluation consists of three types of measurements taken on the evaluation host:

- First, the CPU consumption of the unmodified upstream OVS kernel module was measured during the different levels of throughput
- Second, the modified OVS kernel was installed and its CPU consumption measured while the `int_transit` action was *not* triggered by the generated traffic
- Lastly, the `int_transit` action was enabled and the CPU overhead of the OVS kernel module measured

The experiments resulted no measurable CPU overhead when not triggering the `int_transit` action, which means that the virtual switch performance is not affected by the extension when not using the INT functionality. When enabling the `int_transit` action, the CPU overhead of the kernel module was measured around 0.3% for 1MByte/s and 1% for 100MByte/s for the commodity hardware CPU. The overhead scaled sub-linearly for the intermediate throughputs. These results show that the proposed implementation of an INT capable virtual switch are feasible for practical applications due to low implementation and performance overhead.

V. CONCLUSION

This paper presents a prototypical implementation of the In-Band Network Telemetry Mechanism in the open source software switch Open vSwitch. The presented implementation approach An experimental evaluation of the CPU overhead of the extended OVS kernel module showed that the low resource consumption of the implementation makes it feasible for practical applications.

In the future we would like to further elaborate the approach by implementing additional transport mechanisms like VXLAN or GRE, and extending the experimental evaluation of the performance overhead by more details regarding the CPU and memory consumption and added packet processing latency.

REFERENCES

[1] A. Gulenko, M. Wallschläger, F. Schmidt, O. Kao, and F. Liu, "A System Architecture for Real-Time Anomaly Detection in Large-Scale NFV Systems," *Procedia Computer Science*, vol. 94, pp. 491–496, 2016.

[2] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-Band Network Telemetry via Programmable Dataplanes," in *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2015.

[3] C. Kim, P. Bhide, E. Doe, H. Holbrook, A. Ghanwani, D. Daly, M. Hira, and B. Davie, "Inband Network Telemetry (INT)," 2016. Available online: https://p4.org/assets/INT-current-spec.pdf (Accessed 01-06-2018).

[4] F. Maino, L. Kreeger, and U. Elzur, "Generic Protocol Extension for VXLAN," Internet-Draft draft-ietf-nvo3-vxlan-gpe-06, Internet Engineering Task Force, Apr. 2018. Work in Progress.

[5] A. Begel, S. McCanne, and S. L. Graham, "BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 29, pp. 123–134, ACM, 1999.

[6] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture.," in *USENIX winter*, vol. 93, 1993.

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

[8] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "Pisces: A Programmable, Protocol-Independent Software Switch," in *SIGCOMM Conference*, pp. 525–538, ACM, 2016.