Towards Cross-Layer Telemetry

Justin Iurman
Université de Liège, Montefiore Institute
Belgium
justin.iurman@uliege.be

Frank Brockners
Cisco
Germany
fbrockne@cisco.com

Benoit Donnet
Université de Liège, Montefiore Institute
Belgium
benoit.donnet@uliege.be

ABSTRACT

This paper introduces *Cross-Layer Telemetry* (Clt), a way to combine in-band telemetry (based on In-Situ Oam) and Application Performance Management (APM, based on distributed tracing) into a single monitoring tool providing a full network stack observability. Using Clt, APM traces are correlated with network telemetry information, providing a better view and faster root cause analysis in case of issue. In this paper, we describe the Clt implementation and discuss a use case demonstrating its efficiency. All Clt source code is available as open source.

CCS CONCEPTS

• **Networks** → **Network measurement**; **Network manageability**; **Network monitoring**.

KEYWORDS

Clt, Ioam, OpenTelemetry, Cross-Layer, Telemetry, APM, Jaeger

1 INTRODUCTION

The last decade has witnessed a strong evolution of the Internet: from a hierarchical, relatively sparsely interconnected network to a flatter and much more densely inter-connected network [5, 11, 27] in which hyper giant distribution networks (HGDNs, - e.g., Facebook, Google, Netflix) are responsible for a large portion of the world traffic [2]. HGDNs are becoming the de-facto main actors of the modern Internet. The very same set of actors have fueled the move to very large data center networks (DCNs), along with the evolution to cloud native networking.

ANRW '21, July 24–30, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8618-0/21/07…$15.00
https://doi.org/10.1145/3472305.3472313

Throughout the years, multiple *Operations, Administration, and Maintenance* (Oam) tools have been developed, for various layers in the protocol stack [23], going from basic `traceroute` to Bidirectional Forwarding Detection (BFD [22]) or recent UdpPinger [10] and Fbtracert [9]. The measurement techniques developed under the Oam framework have the potential for performing fault detection and isolation and for performance measurements.

Telemetry information (e.g., timestamps, sequence numbers, or even generic data such as queue size, geolocation of the node that forwarded the packet) is key to HGDNs, DCNs, and Internet operators in order to tackle two particular challenges. First, the network infrastructure must be running all the time, even in the presence of (unavoidable) equipment failure, congestion, or change of traffic patterns. Said otherwise, it means that HGDNs and DCNs must carefully engineer their network infrastructure to be able to ensure that issues are responded to within seconds. Network monitoring and measurements are thus of the highest importance for HGDNs and DCNs, though the available tools and methods [9, 10] have not kept up with the pace of growth in speed and complexity. Second, customers want to enjoy their content in whatever context they access it: at home behind a DSL gateway, on a mobile device in public transportation, at home on multiple devices at the same time, etc. In addition, customers want to experience their content with the highest possible quality and the lowest delay without interfering with the network. Consequently, HGDNs, DCNs, and classical Internet operators must carefully engineer their network to ensure the highest Quality-of-Experience (QoE) on the user side, especially with the emergence of *microservices*.

Modern cloud-native applications rely on *microservices*, namely independent services providing a specific core function. A single request in an application can invoke a lot of *microservices* interacting with each other. As a matter of fact, it is more and more difficult to monitor and isolate a problem, e.g., a slowdown of a service. This is why Application Performance Management (APM, based on distributed tracing tools like `OpenTelemetry` [25] or `Jaeger` [21]) is useful. It provides a way to observe and understand a whole chain of events in a complex interaction between *microservices*. However, such APM appears as useless when the problem is not application related but rather located at the network level.

To solve such a problem, this paper introduces *Cross-Layer Telemetry* (Clt), i.e., device level, flow level, packet level, and application telemetry at the same time. Clt combines APM with network telemetry as provided by In-Situ Oam (Ioam [3]). In a nutshell, Ioam gathers telemetry and operational information along a path, within packets, as part of an existing (possibly additional) header. It is encapsulated in Ipv6 packets as an *Ipv6 HopByHop extension header* [1, 4]. The purpose of APM is to capture and export data from cloud native applications, to receive tracing telemetry data and to provide processing, aggregating, data mining, and visualizations of that data.

From the HGDNs and DCNs perspective, Clt offers an integrated view of the network (APM traces are correlated with network telemetry information), leading so to a careful and efficient integrated network monitoring. Further, if partial Clt information is embedded in data packets (thanks to Ioam) rather than being sent within probe packets (e.g., `ping`, `traceroute`), it has the potential to reduce the telemetry traffic, avoiding so to burden the network and letting the network carrying data traffic instead of monitoring one.

The remainder of this paper is organized as follows: Sec. 2 describes our Clt implementation and associated challenges; Sec. 3 demonstrates Clt efficiency through a real world use case; Sec. 4 discusses next steps for Clt being largely deployed; finally, Sec. 5 concludes this paper by summarizing its main achievements.

2 CROSS-LAYER TELEMETRY IMPLEMENTATION

Let us assume an HTTPS request to be monitored. With APM (e.g., `OpenTelemetry` [25] or `Jaeger` [21]), one obtains useful information on the application level based on application traces (i.e., L5 → L7). However, picture now a situation in which one notices an abnormal execution time (e.g., too long). With APM, it is impossible to exactly understand why it happens. Worst, if the problem is not application related but, rather, on the link or on intermediate hops (e.g., due to congestion), one will be stuck wondering why the request takes so long as the application side looks fine. A better solution would be to show how the request progresses hop-by-hop through the network and identify (potential) bottlenecks. Indeed, by correlating network level telemetry (i.e., network packets) with APM traces, one would give operators a far more complete tool to deal with problems. This is exactly what we want to achieve with *Cross-Layer Telemetry* (Clt), as it makes the entire network stack (i.e., from L2 → L7) visible to monitoring tools, instead of the classic application level visibility. This section discusses our Clt implementation.

Clt relies on Ioam (Sec. 2.1) for network level telemetry (L2 → L4) and on a distributed tracing tool for APM (L5 → L7 – Sec. 2.2). For this implementation, we choose to use `Jaeger` as the tracing tool, although Clt is generic enough to integrate any other alternative to `Jaeger`. The matching between both is achieved by extending Ioam headers (Sec. 2.3). Finally, a telemetry agent (Sec. 2.4) is responsible for extracting data from traces and sending them to a collector for later processing. The Clt architecture is illustrated in Fig. 1.

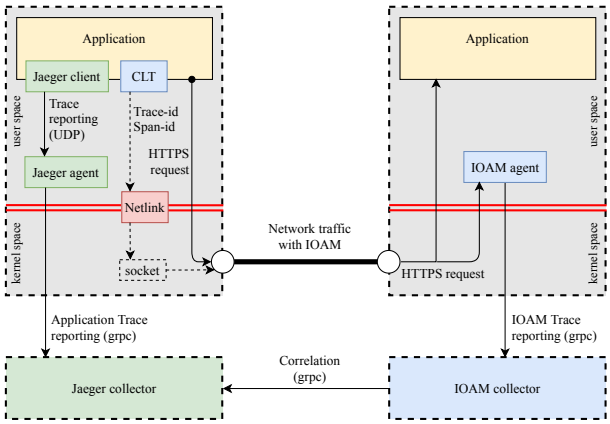

Figure 1: Cross-Layer Telemetry architecture.

2.1 Network Level Telemetry

In-Situ Oam (Ioam) for Ipv6 [1] has been designed for carrying telemetry data within packet headers, for example as part of an Ipv6 Extension Header [4]. Typically, Ioam is deployed in a given domain, between the Ingress and the Egress or between selected devices within the domain. Each node involved in Ioam may insert or update an Ioam header. Ioam data is added to a packet upon entering the domain and is removed from the packet when exiting the domain.

Ioam data fields are associated to *Ioam namespaces*, that are identified by a 16-bit identifier. They allow devices that are Ioam capable to determine whether Ioam option headers need to be processed or updated, and also provide additional context for Ioam data fields. Ioam namespaces can be used by an operator to distinguish different operational domains.

We have implemented Ioam, and more specifically the Ioam Pre-allocated Trace Option (Pto– the space for Ioam data is pre-allocated in the packet header at the Ingress for the Ioam domain), as a patch for the Linux kernel [14, 19]. The Ioam Pto can carry data, such as, e.g., ingress and/or egress interface IDs, timestamps, queue size, buffer occupancy, etc. Our implementation provides data plane support for Ioam, both for processing the header and for the configuration of Ioam namespaces through netlink [24]. It also provides control plane support for Ioam through a netlink interface to configure the Ioam insertion, which is per route configurable. In parallel, we have also implemented the Ioam

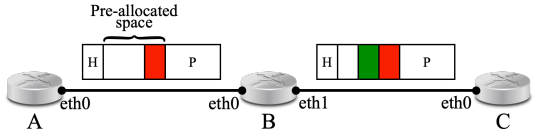

Figure 2: IOAM PTO example. "H" refers to the packet header, while "P" is the payload. Telemetry data (red and green) is inserted in the pre-allocated space. Router *A* is the INGRESS of the IOAM domain, while *C* is the EGRESS.

```
1   $ sysctl -w net.ipv6.conf.eth0.ioam6_enabled=1
2   $ ip ioam namespace add 123
3   $ ip -6 route add db02::/64 encap ioam6 trace type 0x800000 ns 123 size 12
        dev eth0
```

Figure 3: IOAM command-line configuration.

support [17] for `iproute2` that uses the netlink interface provided for the control plane configuration.

Fig. 2 illustrates how the IOAM PTO works. To set it up, the operator must configure an IOAM domain between the three nodes. In this case, IOAM is only used from *A* (INGRESS) to *C* (EGRESS) but not on the reverse path, meaning IOAM must be allowed for both *B.eth*0 and *C.eth*0. This is easily achieved through `sysctl` (see Line 1 on Fig. 3). Then, an IOAM namespace (e.g., ID 123) is created on each node (see Line 2 on Fig. 3). Finally, *A* must be configured to insert an IOAM PTO in its packets when *C* (e.g., `db02::2`) is the destination (see Line 3 on Fig. 3).

As a result, when *C* is the destination, *A* pre-allocates room for the IOAM trace and inserts its IOAM data corresponding to the red block in Fig. 2. IOAM headers are carried inside an IPV6 HopByHop Extension Header, as IPV6 Options. Upon receiving packets with an IOAM PTO, *B* in turn inserts its IOAM data (the green block on Fig. 2). *C* does the same as *B*, but it is not visible as *C* is the destination. In the end, the full IOAM trace is available on *C* for processing.

2.2 Application Performance Management

`Jaeger` is a public and free `APM` tool, based on distributed tracing that follows `OpenTelemetry` standards [25], giving operators the possibility to monitor their microservices and get some profiling data such as operation name, timing, tags, and logs. Distributed tracing relies on two concepts: *traces* and *spans*. A trace "is a data/execution path through the system and can be thought of as a directed acyclic graph of spans" [20]. A span "represents a logical unit of work that has an operation name, a start time of the operation and a duration. Spans may be nested and ordered to model causal relationships" [20]. Fig. 4 illustrates these concepts. For example, span *A* could be an `HTTPS` request, an algorithm that loops over a list, or anything else one wants to monitor in

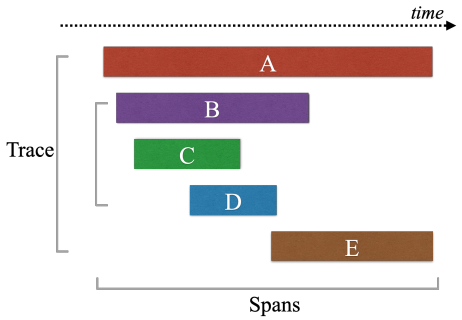

Figure 4: Relationship between a trace and spans.

the application. In this case, the trace represents all spans: $A + B + C + D + E$. Traces and spans are generated within the application by the `Jaeger` client library, according to the monitoring instructions added to the code. Those traces and spans are sent locally over `UDP` to a network daemon, the `Jaeger` agent, that batches and sends them to a remote collector. The `Jaeger` collector receives traces from `Jaeger` agents and runs them through a processing pipeline, i.e., validates traces, indexes them, performs any transformations, and finally stores them [20]. At the end, traces can be retrieved from storage and displayed thanks to a UI service called `Jaeger` Query. The relationship between `Jaeger` components is illustrated by green boxes on the left side of Fig. 1.

To obtain full stack visibility, network telemetry packets must be correlated with `APM` traces. It may appear enough, at first glance, to inject both application trace and span identifiers in the data plane. Unfortunately, a span identifier can vary even within a single `TCP` connection as multiple requests can go over it, meaning it is never going to be a single span identifier per socket. For instance, one could have two `HTTPS` requests to monitor and so two different spans, one for each request. Also, one could use the same socket for all clients and keep it open. Therefore, injecting both trace and span identifiers at socket creation would not be enough. Indeed, the injection must happen at sending time, and so for each request on the socket.

2.3 Full Stack Telemetry

In order to match network level telemetry with `APM` traces, we extend IOAM PTO header with application trace and span identifiers right after the initial header, as illustrated in Fig. 5. Thanks to this enhancement, future correlations on the receiver side between network telemetry information and `APM` traces become possible and easy. With a new patch [15], we reflect the IOAM PTO header modification and provide a way to inject both trace and span identifiers on a socket through netlink. We also provide a CLT client library [15] that encapsulates a netlink call to inject both trace and span identifiers on a socket.

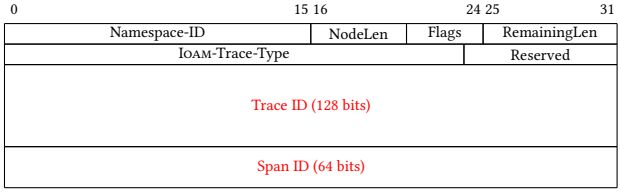

Figure 5: Enhanced IOAM PTO Header.

```
1  span = tracer.start_span('test')
2  CLT.enable(sockfd, span.trace_id, span.span_id)
3  span.start_time = time.time()
4  resp = https.request('GET', '/test') # HTTPS request to monitor
5  span.finish()
6  CLT.disable(sockfd)
```

Figure 6: Example of tracing code with CLT.

An example of code to monitor an HTTPS request with Jaeger is shown in Fig. 6. One can see the difference between the classic solution and the CLT one. With the latter, only two more lines are added to the code, i.e., lines 2 and 6. Further, this technique offers a good "isolation", i.e., IOAM traces are sent and correlated with APM traces only for HTTPS requests and not for other TCP packets (e.g., ACK), which is cleaner on the UI side as it reflects exactly what must be monitored in the code from an application point of view.

2.4 Telemetry Data Collection and Processing

In this telemetry ecosystem, we also provide an IOAM agent [16] to collect and report IOAM traces. Basically, this is a per-interface sniffer for IPv6 packets that filters a *HopByHop* Extension Header containing an IOAM PTO. After parsing, each IOAM trace is represented by our IOAM Trace API [18] defined with Protocol Buffers v3 [12]. The IOAM agent can be run in two different modes: *output* or *report* (default mode). The output mode prints IOAM traces in the command-line interface, while the report mode sends them to a collector through grpc [13].

Finally, we provide an IOAM collector, a Golang interface between the IOAM agent and Jaeger [15]. It enhances a span with IOAM data received from the IOAM agent and reports it to the Jaeger collector. The latter will in turn correlate the classic span with the received-enhanced one.

2.5 Example

Fig. 1 illustrates the interactions between Jaeger and IOAM components in CLT. Based on the code snippet in Fig. 6, let us illustrate what happens step by step. Line 1 uses the Jaeger client library to create a new span called "test" and to add it to the current trace. Line 2 uses the CLT library to inject both trace and span identifiers on the underlying socket through netlink. From now on, any packet with IOAM going out of this socket will include the trace and span identifiers. The only downside is that a packet from a previous trace could be marked with the current one (e.g., due to queue congestion), which is half a problem. Indeed, the main goal is to spot network issues regardless of the connection. However, we are working on a new per packet solution that will provide a real representation of connections. Line 3 defines the start of the span (i.e., the start of the monitoring) by storing the current timestamp. Line 4 executes an HTTPS request. Since the IOAM PTO insertion is configured on the node, IOAM will be included in the network traffic. On the receiver side, the IOAM agent parses and gathers IOAM traces and reports them to the IOAM collector with grpc. Line 5 is reached as soon as the HTTPS request from line 4 is finished, i.e., a response is received. The span is stopped and the monitoring of the HTTPS request is done. The Jaeger client library sends the trace to the Jaeger agent, that in turn sends it to the Jaeger collector for storage. Line 6 finally uses the CLT library again to remove both injected trace and span identifiers from the socket, through a netlink call, providing a better "isolation" as explained previously. The trace representing the HTTPS request monitoring can then be viewed with the Jaeger UI and has now per-hop network telemetry attached.

3 USE CASE

This section covers a common use case that demonstrates how CLT can be used to quickly detect and fix a problem, compared to the classic APM. We first present the use case scenario (Sec. 3.1) and how we implement it (Sec. 3.2). Then, we discuss the results (Sec. 3.3), also including the impact of the additional cost caused by CLT usage.

3.1 Scenario

Picture a situation where clients use a mobile application requiring authentication. Therefore, the application sends an HTTPS request towards the corresponding remote API, with the username and password entered by the client. The receiving API entry point hides the business logic behind each request. In this case, a sub-request to authenticate the client is sent to the server. Each sub-request sent by the API entry point is monitored by Jaeger.

Suddenly, huge delays during the login process are reported by multiple users. Consequently, the operator consults the monitoring tool where each result is stored and sees that login traces are showing larger execution times than usual. Surprisingly enough, both the server and its local database look fine at first glance. The operator decides to use CLT, and so enables IOAM on the entry point to attach network telemetry to Jaeger traces.

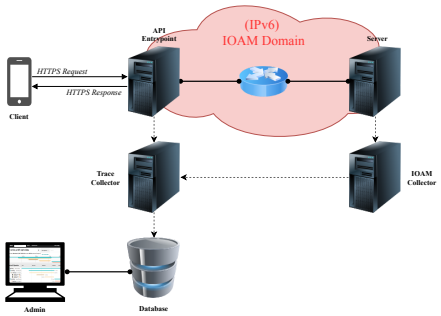

Figure 7: Use case topology

```
$ tc qdisc add dev eth1 root netem delay 100ms
```

Figure 8: `Traffic Control` (tc) command to add a 100ms delay on an interface.

3.2 Testbed

Fig. 7 illustrates the scenario described previously. We use docker [6] to build the topology and docker-compose [7] to ease the configuration between each container. Each topology component is represented by a docker container. The API entry point's application handler uses the `Jaeger` client library to monitor each critical part of the code. It also uses the Clt library to inject both the trace and span identifiers in the underlying socket, which one is kept open for all connections. Still on the entry point, a `Jaeger` agent is running and reporting traces to a collector. The Ioam agent runs on the server and reports every Ioam trace to the Ioam collector. The administrator uses `Jaeger Query` as a web interface to see traces stored in the database in a human-friendly way. Elasticsearch [8] is used as the database to store `Jaeger` traces. Ipv6 is deployed between the API entry point and the server. Ioam is enabled and configured on each node within the domain. The entry point is configured to insert Ioam Pto inside packets when the destination is the server.

In order to simulate a low-level issue, we introduce artificial delay to mimic a congestion on the router between the entry point and the server, thanks to the `Traffic Control` (tc) tool [26]. Fig. 8 shows the command used to add a delay of 100*ms* on the router interface towards the server, which means the RTT will suffer from a 200*ms* increase.

In our experiment, we generate 200 HTTPS requests per second, over a period of four minutes. This time frame is divided in four slices (one minute each): the first minute represents a normal situation where everything runs fine. The congestion (additional 100*ms* delay) is introduced in the second minute. The third minute includes the problem investigation by enabling Clt. Finally, the last minute represents the come back to a normal situation, after the problem has been fixed by the operator. During the four minutes experiment, we measure the RTT of each HTTPS request.

When enabling the Ioam Pto insertion on the API entry point (i.e., enabling Clt), the operator requires the following Ioam data to be included in the trace: the hop limit, the Ioam node-id, both Ingress-id and Egress-id, and the Egress queue depth. Indeed, the latter is included because the operator suspects a congestion somewhere on the path. Of course, additional Ioam data could be required to cover and detect more problems when one has no clue of the issue.

3.3 Results

Fig. 9 shows a screenshot of the `Jaeger UI` with a span representing a login request (*span_login*) that was randomly selected among all login requests during the third minute of the experiment (i.e., Clt enabled). The *ioam_span* is the enhanced span attached to the classic one. Thanks to the latter, the operator quickly detects that the Egress queue of the router is increasing (see the red rectangle), meaning there is a congestion. An action can then be applied to fix the problem, e.g., by re-balancing traffic over queues. Without Clt, the operator would have faced a lot more difficulties in performing root cause analysis.

The experiment performed also allows us to measure the impact of Clt (i.e., the injection of trace and span identifiers on the socket through netlink and the Ioam Pto header insertion). Fig. 10 shows the RTT measured during the four minutes experiment. During the second minute, one can clearly see that the RTT has increased by 200*ms* due to the simulated congestion. The key part is the third minute, during which Clt is enabled. Indeed, the distribution on the graph looks the same for both the second and third minute, demonstrating so the Clt lightness. In order to make sure that Clt is really efficient, we also perform a similar experiment in three steps without the congestion. The objective is to see how Clt behaves without congestion and so with more traffic. Fig. 11 shows the result. Again, one can see that the distribution on the graph looks the same, both with and without Clt.

Therefore, one can say that Clt is efficient since the introduced overhead is only a Netlink call. The major overhead is due to Ioam and was studied previously [19]. Clt's major improvement is a huge gain of time, as well as a more complete tool, for operators to detect low-level issues that are not application related. It is also worth mentioning that the Clt solution is generic enough to integrate other alternatives to `Jaeger`. Indeed, only the Ioam collector is dependent on the tracing tool and would need a few modifications, which is not the case for both the Clt client library nor the Ioam agent. In the long-term, a solution where both the detection and reaction steps become automatic is a goal to be reached.

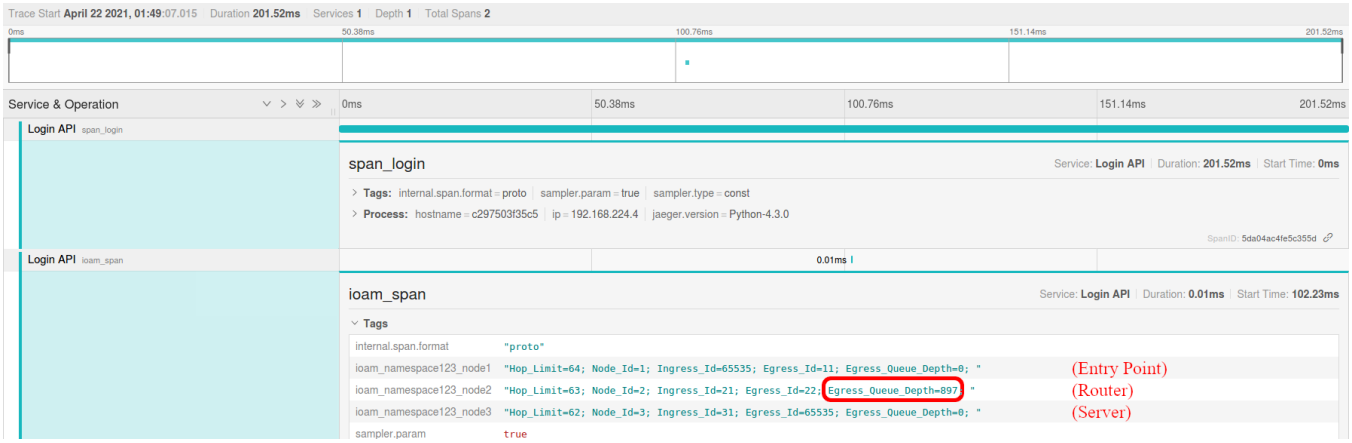

Figure 9: Enhanced span (with IOAM) stored by Jaeger.

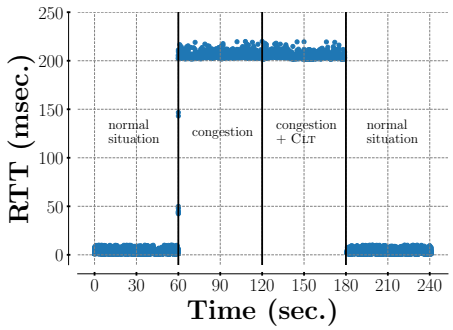

Figure 10: RTT measurement, 200 requests/second, four steps with congestion.

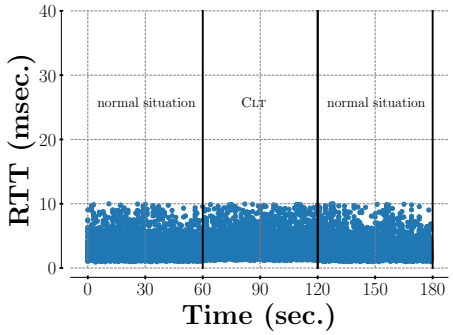

Figure 11: RTT measurement, 200 requests/second, three steps without congestion.

4 NEXT STEPS

In the near future, CLT will be pushed further. First, we plan to standardize the required fields for CLT within IOAM. Then, we plan to standardize IOAM within `OpenTelemetry API`, therefore allowing network telemetry to have its own "stack" in all distributed tracing tools. Depending on how the `OpenTelemetry` standardization will be designed, the current spans hierarchy may need to be rethought. Indeed, what we could provide for, e.g., an `HTTPS` session is a set of child spans showing the individual hops that the `HTTPS` session rides across. For now, the current solution does not require such a modification since the entire IOAM trace is attached to a single span. Finally, we will explore a bit more what additional possibilities CLT enables. For example, CLT could be used in a closed loop context to automatically respond to issues found in the full stack.

5 CONCLUSION

This paper introduced *Cross-Layer Telemetry* (CLT), a new and efficient solution to enhance distributed tracing tools, such as `Jaeger` or `OpenTelemetry`, by correlating Application Performance Management (`APM`) traces with network telemetry information. CLT leverages In-Situ OAM (IOAM) to make the entire network stack (L2 → L7) visible for distributed tools, instead of the classic application level visibility. We do believe that CLT solves challenges from the microservice tracing world and brings a more complete tracing solution to operators to solve lower level issues that are not necessarily application related.

SOFTWARE ARTEFACTS

All the source code required for the CLT implementation as described in this paper is freely available here: https://people.montefiore.uliege.be/bdonnet/telemetry

ACKNOWLEDGMENTS

Mr. Iurman's work has been funded by a Cisco grant CG# 1673376.

REFERENCES

[1] S. Bhandari, F. Brockners, C. Pignataro, H. Gredler, J. Leddy, S. Youell, T. Mizrahi, A. Kfir, B. Gafni, P. Lapukhov, M. Spiegel, S. Krishnan, R. Asati, and M. Smith. 2021. *In-situ OAM IPv6 Options.* Internet Draft (Work in Progress) draft-ietf-ippm-ioam-ipv6-options-05. Internet Engineering Task Force.

[2] T. Böttger, F. Cuadrado, G. Tyson, I. Castro, and S. Uhlig. 2018. Open Connect Everywhere: A Glimpse at the Internet Ecosystem Through the Lens of the Netflix CDN. *ACM SIGCOMM Computer Communication Review* 48, 1 (January 2018).

[3] F. Brockners, S. Bhandari, and T. Mizrahi. 2021. *Data Fields for In-Situ OAM.* Internet Draft (Work in Progress) draft-ietf-ippm-ioam-data-12. Internet Engineering Task Force.

[4] S. Deering and R. Hinden. 2017. *Internet Protocol, Version 6 (IPv6) Specification.* RFC 8200. Internet Engineering Task Force.

[5] A. Dhamdhere and C. Dovrolis. 2010. The Internet is Flat: Modeling the Transition from a Transit Hierarchy to a Peering Mesh. In *Proc. ACM CoNEXT.*

[6] Docker. [n.d.]. Empowering App Development for Developers. See https://www.docker.com/.

[7] Docker. [n.d.]. Overview of Docker Compose. See https://docs.docker.com/compose/.

[8] Elastic. [n.d.]. The Official Distributed Search and Analytics Engine. See https://www.elastic.co/elasticsearch/.

[9] Facebook. [n.d.]. fbtracert. See https://github.com/facebook/fbtracert.

[10] Facebook. [n.d.]. UdpPinger. See https://github.com/facebook/UdpPinger.

[11] P. Gill, M. Arlitt, Z. Li, and A. Mahant. 2008. The Flattening Internet Topology: Natural Evolution, Unsightly Barnacles or Contrived Collapse?. In *Proc. Passive and Active Measurement Conference (PAM).*

[12] Google. 2008. Protocol Buffers – Google's data interchange format. See https://github.com/protocolbuffers/protobuf.

[13] grpc. [n.d.]. A High Performance, Open Source Universal RPC Framework. See https://grpc.io.

[14] J. Iurman. 2020. Kernel patch for IPv6 IOAM. See https://github.com/iurmanj/kernel_ipv6_ioam.

[15] J. Iurman. 2021. Cross Layer Telemetry. See https://github.com/iurmanj/cross-layer-telemetry.

[16] J. Iurman. 2021. IOAM Agent for Python3. See https://github.com/iurmanj/ioam-agent.

[17] J. Iurman. 2021. IOAM Patch for iproute2. See https://github.com/iurmanj/kernel_ipv6_ioam/tree/master/iproute2.

[18] J. Iurman. 2021. IOAM Trace API with Protocol Buffers v3. See https://github.com/IurmanJ/ioam-proto3.

[19] J. Iurman, B. Donnet, and F. Brockners. 2020. Implementation of IPv6 IOAM in Linux Kernel. In *Proc. Technical Conference on Linux Networking (Netdev 0x14).*

[20] Jaeger. [n.d.]. Architecture. See https://www.jaegertracing.io/docs/1.22/architecture/.

[21] Jaeger. [n.d.]. Open-Source, End-to-End Distributed Tracing. See https://www.jaegertracing.io.

[22] D. Katz and D. Ward. 2010. *Bidirectional Forwarding Detection (BFD).* RFC 5880. Internet Engineering Task Force.

[23] T. Mizrahi, N. Sprecher, E. Bellagamba, and Y. Weingarten. 2014. *An Overview of Operations, Administration, and Maintenance (OAM) Tools.* RFC 7276. Internet Engineering Task Force.

[24] netlink. [n.d.]. netlink(7) – Linux manual page. See https://man7.org/linux/man-pages/man7/netlink.7.html.

[25] OpenTelemetry. [n.d.]. Effective Observability Requires High-Quality Telemetry. See https://opentelemetry.io.

[26] tc. [n.d.]. tc(8) – Linux manual page. See https://man7.org/linux/man-pages/man8/tc.8.html.

[27] H. Zhao and J. Bi. 2013. Characterizing and Analysis of the Flattening Internet Topology. In *Proc. International Symposium on Computers and Communications (ISCC).*