Cross-Layer Telemetry Support in Linux Kernel

Justin Iurman, Benoit Donnet
Université de Liège, Montefiore Institute – Belgium

September 27, 2022

Abstract

This paper introduces *Cross-Layer Telemetry* (CLT), presents its new version as an improvement and explains how its support is added to the Linux kernel. CLT is a way to combine in-band telemetry and Application Performance Management (`APM`, based on distributed tracing with `OpenTelemetry`) into a single monitoring tool providing a full network stack observability. Using CLT, `APM` traces are correlated with corresponding network traffic, providing a better view and a faster root cause analysis in case of issue. This new version improves the correlation accuracy. In this paper, we describe the CLT implementation (both for kernel and user spaces) and we evaluate the CLT ecosystem based on a use case. All CLT code is available as open source.

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 [11, 5, 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.

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 and 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 [10, 9] 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 *microser-*

vices, 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 following `OpenTelemetry` [25] standards) 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 *`Hop-by-Hop` extension header* [4, 1]. 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.

2 Application Performance Management

`OpenTelemetry` [25] is a public and free `APM` tool, based on distributed tracing, giving operators the possibility to monitor their microservices and get some profiling data such as operation name, timing, tags, and logs. It is also a standard. 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

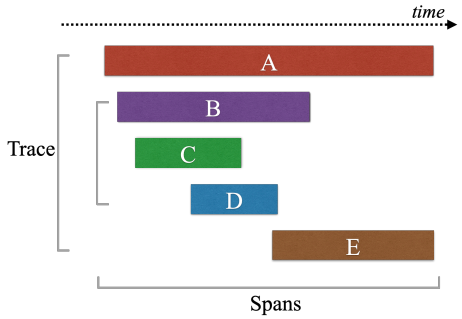

Figure 1: Relationship between a trace and spans.

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. 1 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 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 `OpenTelemetry` client library, according to the monitoring instructions added to the code. Those traces and spans are sent to a remote collector via the `OpenTelemetry` exporter. The back-end collector (e.g., `Jaeger` [21]) receives traces 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 for data visualization.

3 CLT Implementation

Let us assume an `HTTPS` request to be monitored. With `APM` (e.g., distributed tracing tools following `OpenTelemetry` standards), one obtains useful information on the application level based on application traces (i.e., L5 $\rightarrow$ 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 the purpose of *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.

3.1 Clt History

To obtain full stack visibility, network telemetry packets must be correlated with `APM` traces. However, such a correlation based on trace and span ids is not as easy as it seems. 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. Worse, one could use the same socket for all clients and keep it open. In this case, multiple traces would go through the same socket. 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. A natural idea that comes to mind is therefore to overload the existing send system calls. This solution would work but, unfortunately, it would be too disruptive as it would require both `libc` and high-level languages to be modified accordingly. As a consequence, the only viable alternative is to use `netlink` [24] to pass the identifiers to the kernel before sending data.

The first version, let us call it the socket annotation technique, uses a `netlink` call to pass the identifiers to the kernel, which in turn copies them into the corresponding `sock` structure. Each packet going out of this socket would therefore carry the identifiers. Although this solution was working really well, it had a non-negligible downside, namely packets that could be marked with the wrong identifiers due to, e.g.,

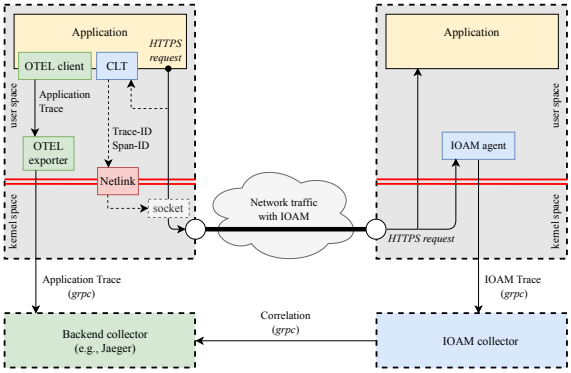

Figure 2: Cross-Layer Telemetry architecture.

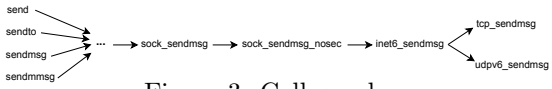

Figure 3: Call graph.

congestion in a queue. Indeed, a packet for the annotation X could still be in the queue when another annotation Y erases the previous one, which would make the packet annotated with Y instead of X as soon as it is sent out.

The second version, let us call it the socket buffer annotation technique (which is the current one), deals with the aforementioned problem. Fig. 2 illustrates its architecture and is explained in Sec. 3.2 and Sec. 3.3.

3.2 Kernel Space

The kernel patch is available on the Clt repository [17] and is quite straightforward. Briefly, we extend both `sock` and `sk_buff` structures with two fields: (*i*) a 128-bit field for the trace id; and (*ii*) a 64-bit field for the span id. A `netlink` call is added to pass identifiers (trace and span IDs) from user space to the kernel right before sending data, and so for a specific socket file descriptor. As for the first version, these identifiers are copied inside the corresponding *sock* structure. But now, as soon as a packet is created to send data through the annotated socket, these identifiers are also copied to the socket buffer (`skb`) itself. Fig. 3 shows the call graph of `send` system calls where we

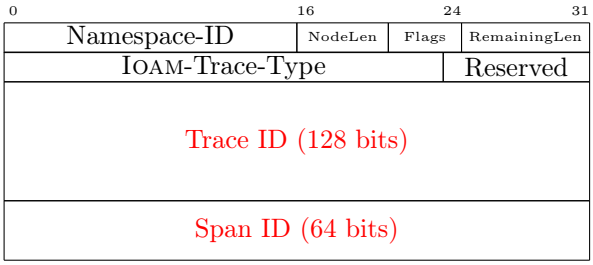

Figure 4: Enhanced IOAM PTO Header.

```
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.end()
6  CLT.disable(sockfd)
```

Figure 5: Example of tracing code with CLT.

respectively identified `tcp_sendmsg` (more precisely `tcp_sendmsg_locked`) for `TCP` and `udpv6_sendmsg` (more precisely `__ip6_make_skb`) for `UDP` to be the two functions that create socket buffers. Thanks to that, we make sure that a socket buffer is annotated directly and correctly, instead of waiting for the packet to be sent out through the socket.

CLT relies on IOAM, in particular on the IOAM Pre-allocated Trace Option-Type (PTO). This option has been designed for carrying telemetry data within a `Hop-by-Hop` Extension Header. 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, and also provide additional context for IOAM data fields. IOAM namespaces can be used by an operator to distinguish different operational domains. Support for the IOAM PTO (the space for IOAM data is pre-allocated in the packet header at the INGRESS for the IOAM domain) is available in the kernel since release 5.15. The IOAM PTO can carry data such as, e.g., ingress and egress interface IDs, timestamps, queue size, buffer occupancy, etc. As such, IOAM PTO is the perfect candidate to embed both span and trace IDs as it not only carries both identifiers by extending the header, but it also collects telemetry data at the same time, and so for each hop on the path. Fig. 4 illustrates how IOAM PTO header is extended to support span and trace IDs.

3.3 User Space

A CLT client library is provided [17] for user space. It is only available for python clients at the moment, but could easily be implemented for other languages depending on the needs. The CLT client library encapsulates the new `netlink` call logic to pass identifiers to the kernel. An example of code to monitor an `HTTPS` request with `OpenTelemetry` is shown in Fig. 5. One can see the difference between the classic solution and the CLT one. With the latter, only two additional code lines are required, i.e., lines 2 and 6. Further, this technique offers a good "isolation". Indeed, IOAM traces are included and correlated with `APM` traces only for what should be monitored (i.e., an `HTTPS` request in this example, not any other `TCP` packets such as an `ACK`), which is cleaner on the `UI` side as it exactly reflects the monitored block of code from an application point of view. The only downside is that most programming languages do not easily expose the socket layer from the application layer, for example with an `HTTPS` object to send requests, due to how implementation is layered. Therefore, in the application code, the socket must be manually injected inside such an object, or the opposite to get back the socket file descriptor used by such object. This hack seems easier for some languages than for others, though.

As previously mentioned, CLT relies on IOAM PTO to carry both identifiers along with telemetry data. Therefore, it must be configured with the `iproute2`

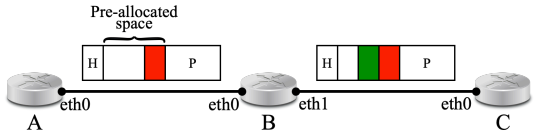

Figure 6: IOAM PTO example. "H" refers to the packet header, while "P" is the payload. Telemetry data (red and green) is inserted in the pre-allocated space. Router A is the INGRESS of the IOAM domain, while C is the EGRESS.

```
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
       prealloc type 0x800000 ns 123 size 12 dev eth0
```

Figure 7: IOAM command-line configuration.

tool that has IOAM PTO support. Fig. 6 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 $B.eth0$ and $C.eth0$. This is easily achieved through `sysctl` (see Line 1 on Fig. 7). Then, an IOAM namespace (e.g., ID 123) is created on each node (see Line 2 on Fig. 7). 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. 7). 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. 6. IOAM PTO is carried inside an IPV6 `Hop-by-Hop` Extension Header. Upon receiving packets with an IOAM PTO, B in turn inserts its IOAM data (the green block on Fig. 6). 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.

3.4 Telemetry Data Collection and Processing

In this telemetry ecosystem, we also provide an IOAM agent [15] to collect and report IOAM traces. Basically, this is a per-interface sniffer for IPV6 packets that filters a `Hop-by-Hop` Extension Header containing an IOAM PTO. After parsing, each IOAM trace is represented by the 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 [19], a Golang interface between the IOAM agent and a back-end collector (in this case, `Jaeger`). 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. Note that the IOAM collector could be implemented in other languages depending on the needs, as well as for other back-ends than `Jaeger`.

3.5 Summary

Fig. 2 illustrates the interactions between each component in CLT. Based on the code snippet in Fig. 5, let us illustrate what happens step by step. Line 1 uses the `OpenTelemetry` 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 created for this socket will include the trace and span identifiers. Line 3 defines the start of the span by storing the current timestamp. Line 4 executes an `HTTPS` request (the instruction to be monitored). Since the IOAM PTO insertion is configured, 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 `OpenTelemetry` client library sends the trace to the `OpenTelemetry` exporter, that in turn sends it to the back-end collector (here, `Jaeger`) for storage. Line 6 clears the effect of Line 2. The trace representing the `HTTPS` request can be observed with the `Jaeger` UI and now contains per-hop telemetry data.

4 Evaluation

This section covers the evaluation of CLT based on a use case described in Sec. 4.1. Then, we discuss the results in Sec. 4.2 as well as the impact of the additional cost introduced by CLT.

4.1 Methodology

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 a server. Each sub-request sent by the API entry point is monitored by `OpenTelemetry`, and `Jaeger` with ElasticSearch [8] as the back-end.

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 after a quick investigation, 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 `OpenTelemetry` traces.

Fig. 8 illustrates the scenario previously described. 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 `OpenTelemetry` 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, the `OpenTelemetry` exporter reports traces to a back-end collector (here, `Jaeger` collector). The IOAM agent runs on the server and reports every IOAM trace to the IOAM collector. The administrator uses `Jaeger` as a web interface to observe traces stored in the database. Elasticsearch is used as the database to store `Jaeger` traces. IPV6 is deployed between the

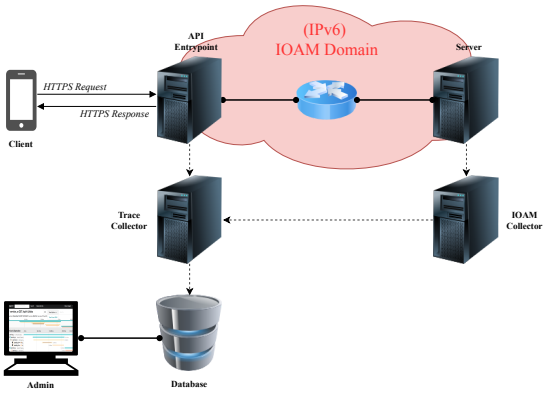

Figure 8: Use case example

```
$ tc qdisc add dev eth1 root netem delay 150ms
```

Figure 9: `Traffic Control` (tc) command to add a 150ms delay on an interface.

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. This reproducible use case is available on the CLT repository [17].

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. 9 shows the command used to add a delay of $150ms$ on the router interface towards the server, which means the `RTT` will suffer from a $300ms$ 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 $150ms$ 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.

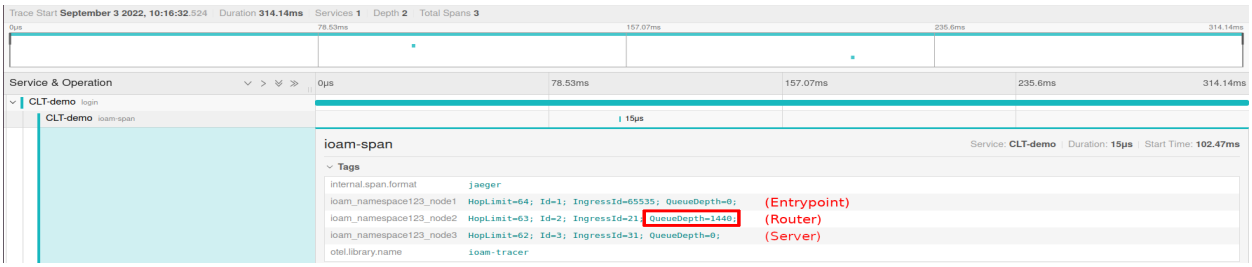

Figure 10: Enhanced span (with IOAM) stored by `Jaeger`.

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 IOAM node-id, both INGRESS id and EGRESS interface IDs, 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.

4.2 Results

Fig. 10 shows a screenshot of the `Jaeger UI` with a span representing a login request 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. 11 shows the RTT measured during the four minutes experiment. During the second minute, one can clearly see that the RTT has increased by 300ms 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

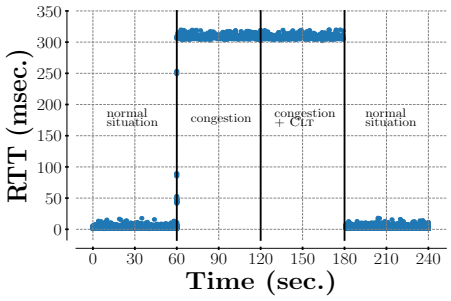

Figure 11: RTT measurement, 200 requests/second, four steps with congestion.

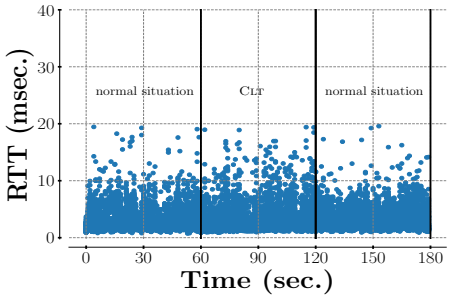

Figure 12: RTT measurement, 200 requests/second, three steps without congestion.

minute, demonstrating that CLT has little impact. In order to make sure that CLT is really efficient, we also perform a similar experiment in three steps without congestion. The objective is to see how CLT behaves without congestion and so with more traffic. Fig. 12 shows the result. Again, one can see that the distribution on the graph looks nearly the same, both with and without CLT.

Therefore, one can say that CLT is quite efficient since the introduced overhead is almost just a netlink call. The major overhead is due to IOAM and was previously studied [14]. The benefits of CLT for operators are a huge gain of time and a more complete tool 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 back-end tool and would need a few modifications, which is not the case for both the CLT client library nor the IOAM agent.

5 Standardization

As previously mentioned, CLT relies on IOAM to kill two birds with one stone: (*i*) IOAM provides per-hop telemetry data and (*ii*) IOAM carries `APM` trace and span identifiers. The latter is achieved by extending the IOAM PTO header. While it might be perfectly fine for a proof-of-concept, it would require something less *hacky* to go through standardization and be widely adopted. A recent draft [16] proposes a way to carry generic identifiers in IPV6 packets, which could be a perfect fit for CLT. Indeed, the correlation between `APM` traces and network traffic would not be tied to IOAM anymore. One could use IOAM alone, or the correlation alone, or both at the same time (two distinct options in a `Hop-by-Hop` Extension Header). In this context, a solution with both at the same time seems obviously better in most cases, as it provides more data and so more chances to spot an issue. But still, one might only want the correlation without per-hop telemetry data, for whatever reason. And, even in that specific case without IOAM for whatever reason not to use IOAM, one could end up with something equivalent to IOAM by running an agent on each hop and report correlations that would include some local, similar-to-IOAM, metrics.

From a kernel point of view, it would mean inserting the aforementioned CLT Option in every IPV6 packets (inside the `Hop-by-Hop` Extension Header), as soon as CLT is enabled on a specific socket by an application, and so even if other options are already inserted in the `Hop-by-Hop`.

6 Conclusion

This paper introduced *Cross-Layer Telemetry* (CLT), a new and efficient solution to enhance distributed tracing tools based on `OpenTelemetry` standards, by correlating Application Performance Management (`APM`) traces with network telemetry information. This paper also explained how CLT support was added to the Linux kernel. CLT leverages In-Situ OAM (IOAM) to make the entire network stack (L2 $\rightarrow$ 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. Therefore, a kernel adoption of CLT support would be a huge step forward.

Source code

The CLT repository gathers all the source code needed for the implementation described in this paper, as well as an explanation on how it works, and a video demo. Everything is freely available here: `https://github.com/Advanced-Observability/cross-layer-telemetry`

Acknowledgments

This work has been funded by a Cisco grant CG# 2713379 and the CyberExcellence project funded by the Walloon Region, under number 2110186.

References

[1] Bhandari, S., and Brockners, F. 2022. In-situ OAM IPv6 Options. Internet Draft (Work in Progress) draft-ietf-ippm-ioam-ipv6-options, Internet Engineering Task Force.

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

[3] Brockners, F.; Bhandari, S.; and Mizrahi, T. 2022. Data Fields for In Situ Operations, Administration, and Maintenance (IOAM). RFC 9197, Internet Engineering Task Force.

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

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

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

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

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

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

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

[11] Gill, P.; Arlitt, M.; Li, Z.; and Mahant, A. 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. A High Performance, Open Source Universal RPC Framework. See `https://grpc.io`.

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

[15] Iurman, J. 2021. IOAMAgent for Python3. See `https://github.com/Advanced-Observability/ioam-agent-python/tree/clt`.

[16] Iurman, J. 2022a. Carrying a Generic Identifier in IPv6 packets. Internet Draft (Work in Progress) draft-iurman-6man-generic-id-00, Internet Engineering Task Force.

[17] Iurman, J. 2022b. Cross Layer Telemetry. See `https://github.com/Advanced-Observability/cross-layer-telemetry`.

[18] Iurman, J. 2022c. IOAMAPI with Protocol Buffers v3. See `https://github.com/Advanced-Observability/ioam-api/tree/clt`.

[19] Iurman, J. 2022d. IOAMCollector (golang interface for Jaeger). See `https://github.com/Advanced-Observability/ioam-collector-go-jaeger`.

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

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

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

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

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

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

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

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