October 2022

PACKET TRACING IN DISTRIBUTED NETWORK ARCHITECTURES

Gaganjeet Reen

Naveen Gujje

Maneesh Soni

Kaarthik Sivakumar

PACKET TRACING IN DISTRIBUTED NETWORK ARCHITECTURES

AUTHORS:
Gaganjeet Reen
Naveen Gujje
Maneesh Soni
Kaarthik Sivakumar

ABSTRACT

In a distributed network architecture, tracking the network function node instances through which a network packet traverses, the time that a packet spends in each of the network functions, and the operations that the functions perform on each packet are extremely important when troubleshooting packet processing issues. However, such distributed architectures often include legacy services which cannot be readily extended (either because of the architecture or because of the programming language used) to include solutions to facilitate the above-described tracing. Techniques are presented herein that support a novel OpenTelemetry-based packet tracing approach that facilitates data plane debugging in a distributed microservices architecture containing heterogenous services without an explicit need for services to integrate with an OpenTelemetry software development kit (SDK). Aspects of the presented techniques encompass the enrichment of log files, a filtering capability, the extension of a Radioactive Tracing-style capability to a microservices world, etc.

DETAILED DESCRIPTION

In a distributed network architecture, network packets pass through several network functions (e.g., microservices) which perform operations on those packets. Tracking the instances through which a packet traverses, the time that a packet spends in each of the network functions, and the operations that the functions perform on each packet are extremely important when troubleshooting packet processing issues. Such distributed architectures often include legacy services that have been lifted and shifted into the cloud and which cannot be readily extended (either because of the architecture or because of the programming language used) to include solutions such as OpenTelemetry to facilitate the above-described tracing.

In short, introducing a packet tracing capability into legacy services, with minimal code changes, is a challenging task.

While several attempts have been made to troubleshoot data plane issues in distributed network systems and service function chains, each of those approaches entails one or more limitations.

A first approach encompasses the use of `traceroute` functionality for troubleshooting service function chains (see, for example, the Internet Engineering Task Force (IETF) Internet-Draft draft-penno-sfc-trace-03). The limitations of this approach are that it requires microservices and service functions to respond to specific trace packets and it does not allow for the troubleshooting of the actual traffic passing through a distributed system. A second approach encompasses a Service Function Chaining (SFC) Path Tracer. The limitation of this approach is that while it allows for the path tracing of simulated traffic, it does not allow for the debugging of data plane issues that might arise within services with live traffic. A third approach encompasses the use of an 'AuditBox.' However, this approach has similar limitations to the previous approach. While it allows for the validating of service function chain characteristics and path information, it does not provide visibility into the operations that each service might perform on a particular packet.

Techniques are presented herein that address the above-described challenge. Aspects of the techniques presented herein leverage elements of the OpenTelemetry initiative.

OpenTelemetry is a well-known open-source framework that supports tracing application-level requests in a distributed environment. Traditionally, services and systems emit OpenTelemetry spans either directly to a central collector or to a local agent which, in turn, forwards those requests to a central collector.

However, such an approach requires incorporating OpenTelemetry libraries into the services, which is not feasible if (as noted previously) there are legacy services in a distributed system or if the services are developed in a programming language that does not have a supported OpenTelemetry software development kit (SDK).

The techniques presented herein facilitate tracing in such heterogeneous distributed environments. Additionally, aspects of the presented techniques trace individual network packets as opposed to most OpenTelemetry-based solutions which trace application-level requests. The techniques presented herein encompass multiple aspects, each of which will be described below.

A first aspect incorporates an open source OpenTelemetry schema that may be used for visualization. The OpenTelemetry concepts, span and trace, are utilized to generate a directed acyclic graph (DAG) of a packet's path as the packet traverses through microservices in a distributed system. A span is the primary building block of a distributed trace, representing an individual unit of work that is completed in a distributed system. Each component of a distributed system contributes a span – i.e., a named, timed operation representing a piece of a workflow.

Spans can (and generally do) contain references to other spans, which allows multiple spans to be assembled into one complete trace – i.e., a visualization of the life of a request as it moves through a distributed system.

According to the techniques presented herein, a trace (which may be identified by a unique trace identifier (ID)) represents all of the operations that are performed on a specific packet by all of the services. A span (which may be identified by a unique ID within a trace) encapsulates all of the operations that are performed on a specific packet in a single microservice. Fields in the network packets may be employed to propagate a trace ID and a span ID from one microservice to the next. Whenever a microservice performs any operation on a packet and emits logs, aspects of the techniques presented herein enrich the existing logs with additional information (such as a trace ID, a previous span ID, a current span ID, and a timestamp).

A second aspect of the techniques presented herein encompasses the construction of an extensible infrastructure that utilizes the OpenTelemetry protocol to send data. The enriched logs from different services, from a microservice infrastructure layer, from an orchestration layer, and from any other components may, through the use of a log forwarder, be forwarded to a central location. There, processing logic may be employed to collate the logs, sort them by trace ID and span ID, and convert them into a standard, interoperable, and extensible format. There are many such formats that may be used, OpenTelemetry being a popular one. The newly formatted data may then be sent to a collector or visualizer.

Key during the above-described processing is the taking of logs from legacy services that do not know about or do not use standard formats (such as OpenTelemetry or other modern logging frameworks) and then the programmatic insertion into those artifacts of modern infrastructure capabilities. Instead of requiring services to integrate with OpenTelemetry libraries and exporting spans directly to agents and collectors, aspects of the techniques presented herein only require that services emit their existing logs which are then enriched to allow for the tracing of network packets. Additionally, the out-of-band export of traces also helps in reducing the overhead costs that are involved during the actual traversal of packets.

A third aspect of the techniques presented herein encompasses the construction of an infrastructure that has filtering capabilities to identify the specific packets that need to be traced. The mechanism to identify and track packets for tracing follows what is commonly referred to as Radioactive Tracing. A Radioactive Tracing infrastructure is typically applied to modules within a single system running on a single piece of hardware. Aspects of the techniques presented herein extend this feature to the microservices world.

A filter may be applied on the point-of-entry to a microservices world. Such a filter may be based on the five tuple flow information which implies that it is also possible to filter encrypted traffic. The filtering capabilities may be further extended to include information such as the identity of the user sending a flow, an application type, or any other criteria depending upon the traffic inspection capabilities of the services in a distributed network system.

Once a filter identifies that a packet is to be traced, it may note the information internally in a cache so that subsequent packets for the same flow do not need pass through the filters again, thus helping with performance.

An ingress node may also set a bit in the header of a packet (either directly in the packet, in an overlay header, or through other methods). The rest of the distributed system, the services, as well as the infrastructure components may then use that bit to identify if the packet is to be traced in each of the services through which the packet passes. The egress node of the distributed system may remove the flag prior to letting the packet exit the entire system.

In such a way, all of the services and a microservices infrastructure do not need to be explicitly configured to trace a specific packet based on criteria that a customer cares about. In particular, since different services may know different details about different parts of a packet or a flow, it would be impossible to create a single filter configuration that could be directly applied to all of the services. By applying the filtering criteria at the point of ingress, it is possible to make the services function without understanding the complexity of the filtering criteria.

A fourth aspect of the techniques presented herein encompasses the fact that packet flows in both directions may be traced. The third aspect (which was presented above) described a filtering process and tracing in one direction. For a reverse flow, a similar approach may be established or a cache may be used to identify whether or not a packet must be traced based on the forward path of the packet.

A fifth aspect of the techniques presented herein encompasses the fact that services will automatically scale (or auto-scale) up or down based on the volume of traffic that is flowing through a network. The techniques presented herein are ideal for such auto-scale scenarios. Since the filtering criteria is only applied on ingress, service instances that are launched do not have to learn about any filtering criteria. Instead, they need only look at a packet's Radioactive Tracing bit to see if a packet must be traced within the service. If an instance of a service were to go down (due either to a crash or being auto-scaled down), packets of the same flow will then go to a different instance of the service. That instance of the service does not need to know what criteria must be used for tracing. Instead, it can also track the Radioactive Tracing bit for tracing packets. Since the logs for all of the services and for all of the service instances are centralized on a log server, the failure of a service instance and the migration of the packet processing to a different instance will not in any way impact the ability to troubleshoot the packets of a particular flow. If a service processes a packet and emits a log, that log will be available for analysis.

There is a significant benefit to following the approach that is expressed through the techniques presented herein. For example, through those techniques it becomes possible to bring the logging of all of the existing legacy services to the cloud and microservices world while supporting integrated and unified troubleshooting across all of those services and the infrastructure. Additionally, it becomes possible to track packets across an entire distributed system irrespective of auto-scale up or down being applied on the services. Further, since services employ a Radioactive Tracing bit on a packet’s header to decide whether or not to trace a packet, new instances of services do not need to load any configuration information to determine the filtering criteria.

Figure 1, below, presents elements of an exemplary architecture that is possible according to aspects of the techniques presented herein and that is reflective of the above discussion.

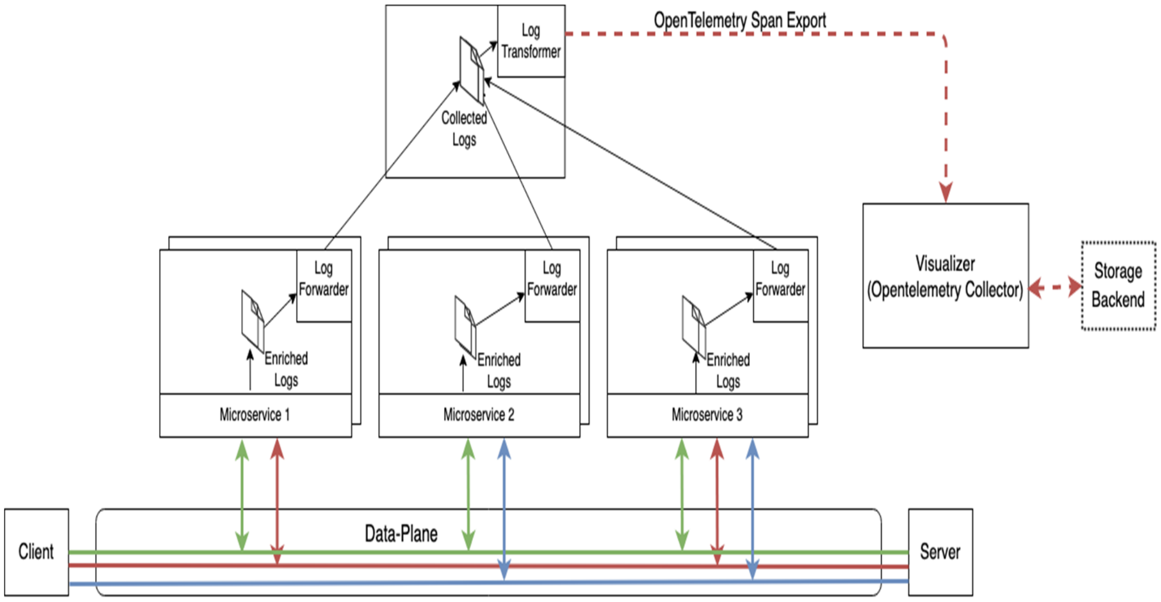


Figure 1: Exemplary Architecture

Figure 2, below, presents elements of a sample packet trace, comprising three spans (corresponding to three services as depicted in Figure 1, above), that is possible according to aspects of the techniques presented herein.

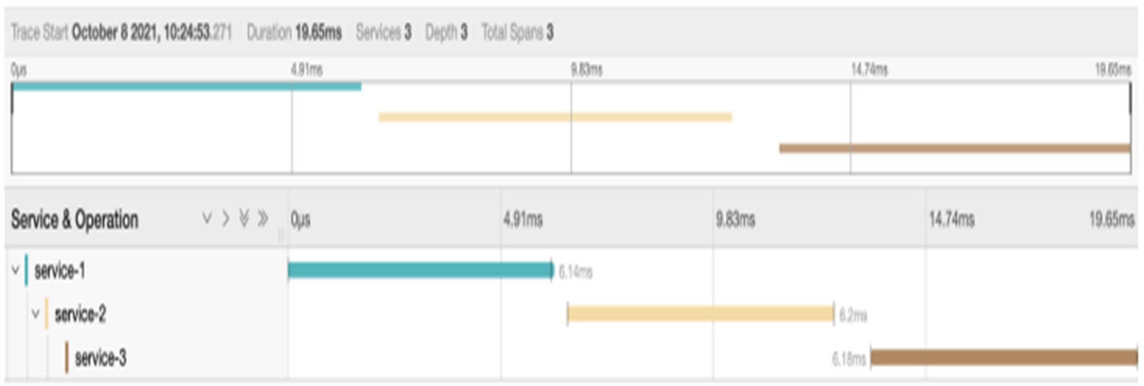
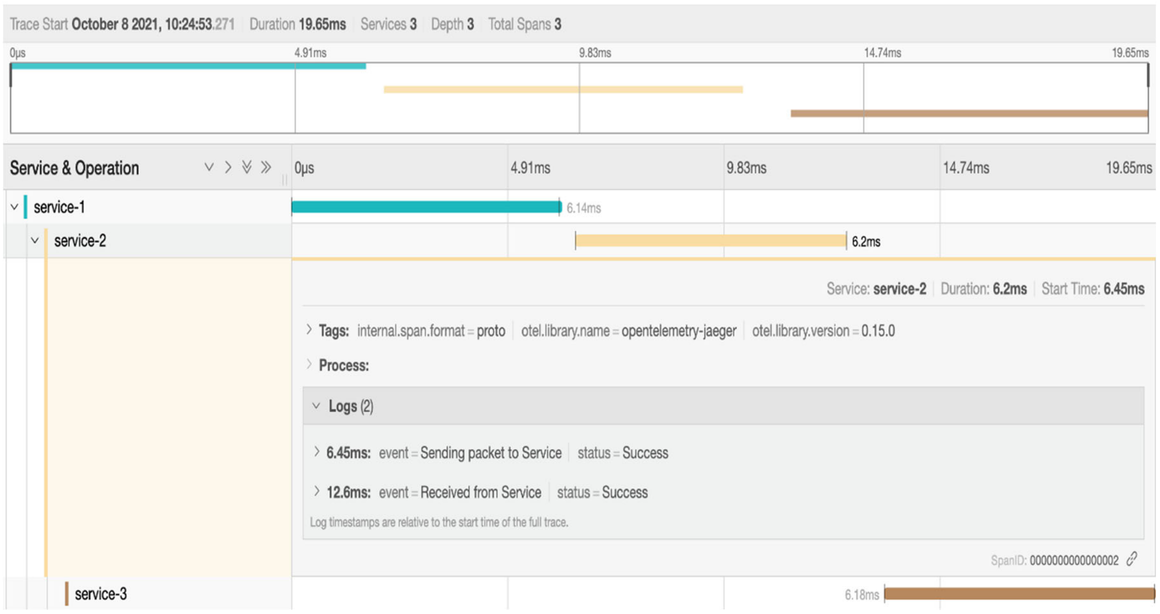


Figure 2: Illustrative Packet Trace

Additionally, Figure 3, below, presents elements of a sample packet trace, comprising logs collected from a service (corresponding to three services as depicted in Figure 2, above), that is possible according to aspects of the techniques presented herein.



Trace Start **October 8 2021, 10:24:53**.271 Duration **19.65ms** Services **3** Depth **3** Total Spans **3**

Service: **service-2** | Duration: **6.2ms** | Start Time: **6.45ms**

Tags: internal.span.format = proto | otel.library.name = opentelemetry-jaeger | otel.library.version = 0.15.0

Process:

Logs (2)

6.45ms: event = Sending packet to Service | status = Success

12.6ms: event = Received from Service | status = Success

Log timestamps are relative to the start time of the full trace.

SpanID: 0000000000000002

Figure 3: Example Logs Collected from Services of Figure 2

In summary, techniques have been presented herein that support a novel OpenTelemetry-based packet tracing approach that facilitates data plane debugging in a distributed microservices architecture containing heterogenous services without an explicit need for services to integrate with an OpenTelemetry SDK. Aspects of the presented techniques encompass the enrichment of log files, a filtering capability, the extension of a Radioactive Tracing-style capability to a microservices world, etc.