Emplacing New Tracing

Adding OpenTelemetry to Envoy

A thesis submitted by

Alexander Ellis

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Tufts University

February 2022

Advisor: Raja R. Sambasivan

Abstract

Distributed tracing is the practice of bringing observability to a microservice-oriented system. It relies on propagating metadata between processes and network boundaries to construct the complete journey of a request through a system, even if that journey requires communication between multiple services. OpenTelemetry, an open-source standard and framework for distributed tracing, has emerged as the front runner standard in distributed tracing in industry. Envoy is a high performance C++ distributed proxy that is commonly used in modern service-mesh architectures. This project focuses on adding OpenTelemetry tracing support to Envoy, allowing for the efficient exporting of OTLP traces from Envoy. This paper explores the design space and decisions for tracing support in Envoy and the OpenTelemetry C++ libraries, and it explores the relationship between general library code and the needs of specialized applications.

Contents

Contents ii

List of Figures iii

List of Tables v

1 Introduction 1

2 Background 2

2.1 Envoy 2

2.2 Distributed Tracing 5

2.3 Envoy's Existing Tracing 6

2.4 Challenges for Integrating OpenTelemetry With Envoy 7

3 Design 8

3.1 Threading 9

3.2 Security and Dependencies 10

3.3 Batching 11

3.4 Connecting to the collector 11

3.5 Future extensibility and the OpenTelemetry C++ API & SDK 12

4 Implementation 13

4.1 Implementation details 13

4.2 Open Source process 14

5 Evaluation 15

5.1 Comparison with Envoy without Tracing 16

5.2 Batching 16

5.3 Connecting to the collector 17

5.4 Comparison with an existing Tracer extension 19

6 Future Work 19

7 Conclusion 21

Acknowledgments 21

References 21

List of Figures

1 From Envoy's perspective, requests flow through the data plane from downstream client to upstream service, while configuration updates follow the control plane from the configuration management server to Envoy. This paper will mostly focus on the data plane. 2

2 Diagram of Envoy's architecture. Requests flow through Envoy from a listening socket through to the Cluster Manager handling upstream connections. 3

3 For a distributed system consisting of microservices, the work done on behalf of a single request can be represented in terms of a trace consisting of temporally and causally ordered spans, each representing the work being done in a service. In the above diagram, for an example Car Rental application, a request from the client to the Car Rental Service spawns a child call to the User Service, which then calls the Auth Service. Following the call to the User Service, the Car Rental Service then contacts the Car Service before returning the response to the client. The collection of spans, representing the entire request, is grouped together as a trace. 5

4 The structure of tracing in Envoy, with the HTTP Connection Manager using a general Tracer API to communicate with framework-specific Tracer implementations. 7

5 Diagram of the worker thread's event loop if we block either waiting for the exporter or during the call to the exporter. In the above example, the Envoy worker thread is asynchronously handling events from two requests, A and B. The sequence is as follows: 1) Envoy receives Request A, 2) Envoy receives request B, 3) Envoy proxies request A upstream, 4) Envoy receives the upstream response for A, 5) Envoy proxies request B upstream, 6) Envoy returns A's response downstream, and 7) Envoy exports A's trace. At this point, if the request blocks, we will not be able to handle any other events while the request is in flight, and accordingly, Envoy can only handle 8) B's upstream response and 9) sending B's response downstream after the exporting call is done, even if they arrived during it. Since the worker thread is a single thread, any blocking that request A does during its life cycle has the potential to block other requests, leading to off-path caused delays. 8

6 Diagram of the architecture of the new OpenTelemetry Tracer. The Tracer is stored in Thread Local Storage, accessible through the Tracer API from HTTP Connection Manager instances on the worker thread. Using Envoy's async gRPC client, the Exporter sends Spans to the Collector through the Cluster Manager's external connection functionality. 13

7 Diagram of the simple test system for evaluation consisting of the Nighthawk Client, Envoy, the Nighthawk Test Server as the single backend, and the OpenTelemetry Collector. 15

8 For a high volume evaluation, changing the batching threshold allowed the OpenTelemetry Collector to "catch up" to the spans coming from the Envoy. 17

List of Tables

1 Some of the design axes for adding the new OpenTelemetry Tracer extension to Envoy. 10

2 Comparing 60 seconds of 1200 RPS being sent to Envoy with and without the new OpenTelemetry Tracer. 17

3 The performance of various batching thresholds and the downstream effect at the Collector.. 18

4 Latencies when comparing 60 seconds of 120 RPS being sent to Envoy with the OpenTelemetry Tracer implemented in three separate ways: with Envoy's asynchronous gRPC client, with a synchronous gRPC client, and with a synchronous gRPC client running in a background thread. 20

5 Latencies from the comparison of batching with Envoy's async gRPC client and batching with the synchronous call in a background thread. 20

6 Latencies from comparing the new Envoy OpenTelemetry Tracer with the Zipkin tracer, which shares some of the design choices. 20

Emplacing New Tracing

Adding OpenTelemetry to Envoy

Alex Ellis[*]

Alexander.Ellis@tufts.com

Tufts University

1 Introduction

Distributed tracing is the practice of bringing observability to a microservice-oriented system. It relies on propagating metadata between processes and network boundaries to construct the complete journey of a request through a system, even if that journey requires communication between multiple services. OpenTelemetry[16], an open-source standard and framework for distributed tracing, has emerged as the front runner standard in distributed tracing in industry. A flexible and vendor-agnostic approach, OpenTelemetry represents both the confluence of different tracing frameworks and the future of open-source tracing. It allows developers an easier and standardized experience for adding observability to their systems.

Envoy[11] is a high performance C++ distributed proxy that is commonly used in modern service-mesh architectures. It provides tracing at the application level, and open source developers have added support for a number of tracing frameworks. There currently is no native OpenTelemetry tracing support in Envoy, and adding support for OpenTelemetry would allow any future users of Envoy to immediately use the new OpenTelemetry Protocol (OTLP) to add observability to their systems with tracing at the proxy-level, and it would move Envoy from the outdated frameworks (OpenTracing and OpenCensus) to the new standard.

Envoy is frequently used in a variety of configurations in a service-mesh microservice architecture, including as an ingress proxy, an egress proxy, a load balancer, or a sidecar; for example, Istio[2] is a popular service mesh framework that uses Envoy as sidecar proxies. Because an Envoy would have full visibility into any work being done as a result of an incoming request, either as a proxy or a sidecar, Envoy represents a natural point to add observability to the system, and instrumenting Envoy allows the user to have a bird's eye view of the work being done in a system. It can create a span for the request that it proxies, representing all work done on behalf of that request. When Envoy is used as a side car proxy for a service, instrumenting Envoy allows for the tracing of all of that service's external requests, even if that service itself has not been instrumented with tracing. With its natural role as a hub for inter-service communications, Envoy can play a key role in revealing the paths through the system. As a first-party OpenTelemetry-traced service, Envoy can mesh with the larger system ecosystem, allowing for a modern and observable system.

This project focuses on adding OpenTelemetry tracing support to Envoy, allowing for the efficient exporting of OTLP traces from Envoy. A second focus is the interface between

[*]This work was performed while the author was employed at Google.

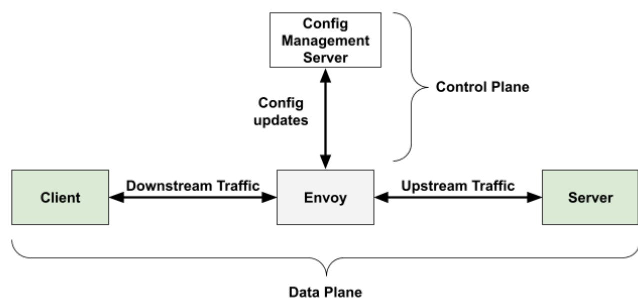


Figure 1. From Envoy's perspective, requests flow through the data plane from downstream client to upstream service, while configuration updates follow the control plane from the configuration management server to Envoy. This paper will mostly focus on the data plane.

tracing and Envoy's industry-level system, specifically the realities of meshing the two projects' goals, architectures, and requirements. This paper explores the design space and decisions for tracing support in Envoy and the OpenTelemetry C++ libraries, and it explores the relationship between general library code and the needs of specialized applications.

2 Background

This section covers the background material for this project, including details on Envoy, distributed tracing, and OpenTelemetry.

2.1 Envoy

Envoy is an open source high performance C++ distributed proxy that is commonly used in modern service-mesh oriented architectures. It is used by many companies, including Lyft, Amazon, Google, Microsoft, Netflix, and many others[11]. Envoy can be used in a variety of ways, including (but not limited to) an ingress proxy, a sidecar for a service, an internal load balancer, and an egress proxy for outgoing traffic.

Envoy is typically situated between a client and one or more backend servers (see Fig. 1). Requests are like salmon; they "swim" upstream from a downstream client to an upstream backend. Though this terminology differs from other uses of the downstream/upstream terms, this paper will use Envoy's terminology. Envoy also allows for dynamic configuration updates via a configuration management server, though the details are outside the scope of this paper. Envoy's surfaces can be separated into the *control plane*, which is the path of configuration updates, and the *data plane*, which is the path requests take through Envoy as they are proxied to upstream services.

Envoy's data plane request handling architecture has two main parts: the listener subsystem, which is responsible for downstream request processing, and the cluster subsystem, which is responsible for selecting and connecting to the upstream connection (see Fig. 2). The steps a request goes through can be thought of roughly as a series of stages from listener to upstream connection pool and back again. Once a request is accepted by a worker thread (see Section 2.1.1), it goes through a series of stages, called *filters*, at multiple abstraction levels that allow Envoy to work on a request at the corresponding level. For example, there could be a listener filter that performs TLS inspection, a Network filter that performs IP-based rate limiting, or an HTTP filter that performs some authorization check on an HTTP header. Developers may write custom filters, and these filters can be executed on the request path, the response path, or in both directions. The filters at each level are collectively called *filter chains*; for example, the group of filters at the L7 level is called the HTTP filter chain.

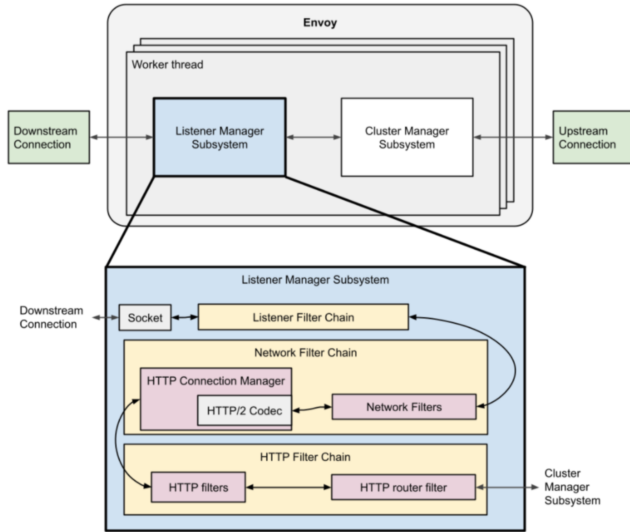


Figure 2. Diagram of Envoy's architecture. Requests flow through Envoy from a listening socket through to the Cluster Manager handling upstream connections.

The final filter in the HTTP filter chain is the HTTP Router Filter, which finalizes route selection and selects an upstream cluster that will be used as the destination for the proxied request. This decision is then passed to the Cluster Manager, which handles selecting the backend for a cluster, load balancing, upstream connection pools, and proxying the request to the upstream service. Once that upstream service has responded, the response flows through the filter chains in the reverse order and is eventually sent back to the downstream connection.

Of particular interest to this paper is the application level. When the request moves from the Network filter chain at the network level to the HTTP filter chain at the application level, it must go through the HTTP/2 Codec to be translated from a byte buffer to a more accessible HTTP representation and vice versa on the response path (see Figure 2). This is done within the HTTP Connection Manager, which is the last step of the network filter chain. It has ownership over the lifecycle of the request at the application level, including statistics and tracing (see Section 2.3).

All filter chains are created and run on demand; for the network filter chain, this would be after the data from the TCP connection has been decrypted, while for the HTTP filter chain, this would be done for each HTTP stream after the HTTP/2 Codec deframes and demultiplexes the incoming data stream. This means that an instance of each filter chain is created for each connection and stream when it is required, and each filter is created uniquely for the part of the request that needs it. For example, for two separate connections A and B, each would have its own instances of the Listener filter chain, the network filter chain, the HTTP filter chain, and importantly for this paper, the HTTP Connection Manager.

2.1.1 Envoy's Threading Model.

Envoy's threading model consists of a single main thread and multiple siloed worker threads, with each thread containing an event loop (via libevent[12]). These worker threads use as little blocking as possible, as a single worker thread may be handling many connections at the same time. This design is similar to that of NGINX[14], another high performance proxy.

One of Envoy's key design principles is reducing shared state and concurrency-based code as much as possible by having independent worker threads working in parallel with limited cross-thread communication. The main thread handles general administrative tasks, such as server startup and shutdown, receiving dynamic config updates via Envoy's various discovery service APIs, process management, and surfacing the administrative interface. The worker threads are responsible for handling connections throughout their entire lifetimes, from listening for new connections to proxying

upstream traffic. Envoy also has additional threads responsible for background work such as file flushing for written local files (such as access logs). By default, Envoy creates a worker thread for each actual hardware thread, and tuning this number is important for striking the balance between idle connections, memory waste, and CPU usage for each thread.

A connection begins with the kernel listening on a given socket. When the kernel accepts a connection, it passes the connection to one of the worker threads listening on that socket. Once a worker thread takes on a connection, the connection will only be on that worker thread throughout the entire connection lifecycle. For each new connection, the worker thread creates the filter chains that it will use and the connection passes through them accordingly with all further processing occuring on that same thread. Each thread runs as a non-blocking event loop via libevent, and each worker thread is able to handle multiple connections with any explicit waiting done via event-based callbacks.

One benefit from this approach is that the majority of the code written for connection handling on each worker thread can be written as if it is a single-threaded application, which greatly reduces the complexity required for both writing and reasoning about the code[18]. The other main benefit is that this approach scales well with the number of worker threads, as the addition of more worker threads does not create additional bottlenecking with any shared resources.

An important implication is that there are multiple connections spending their entire lifecycles on a single worker thread, and if that thread is blocked for any one single connection, any other connections on that thread will also be blocked. This requires some care when writing code on the connection path, as it means that all potentially blocking events should be handled with event-based callbacks, allowing the thread to perform other work in the meantime. For example, if there is an HTTP filter that performs an RPC to an external service for an authorization check for a given request, this RPC must be handled with a callback and not block, as blocking the worker thread would prohibit any other requests from moving forward until the RPC returns. That being said, Envoy does use some blocking behavior when necessary (such as workers acquiring a lock before writing access logs), but these situations are generally low-contention [13].

Envoy has a mechanism for sharing information across threads called thread local storage [13]. The main use is efficient sharing of information from the main thread to the worker threads. One example is passing configuration updates, where the main thread handles configuration changes and posts updates to the worker threads. With thread local storage, one can also store objects in the thread local storage that will be accessible to code inside each worker; for instance, a shared gRPC client stored in thread local storage could be used by any code running inside the thread, though the aforementioned blocking considerations would still apply.

2.1.2 Envoy's Security Model. Envoy's security model assigns safety to upstream or downstream connections for each extension. It has an extensive security review process for external dependencies, and there is an active effort to reduce the dependency on libcurl, which, from a security perspective, has been a particularly problematic library in the past few years.

Envoy's threat model is also divided into data plane and control plane, with the data plane being broken up into downstream and upstream, as visualized in Fig. 1. For the data plane, core and extension components in Envoy can be exposed to downstream and/or upstream traffic, and accordingly, each component has its own security model. Core components can be used with both untrusted and trusted services, while extensions may be hardened against some combination of trusted or untrusted upstreams and downstreams. For example, the Envoy OpenCensus Tracer extension is intended to be hardened against untrusted downstreams, but it assumes that the upstream OpenCensus Collector is trusted.

The main caveat here is that any libraries used in extensions are assumed to be secure, but unfortunately, this is not always the case. For instance, many Envoy extensions use libcurl for performing HTTP requests, including in the AWS extension common utils and the OpenCensus Zipkin Span exporter implementation. libcurl has been responsible for 14 CVEs since 2018[19] (as of 2020) and there is an ongoing effort to remove it as an Envoy dependency[23].

2.2 Distributed Tracing

Distributed tracing is the practice of bringing observability to a microservice-oriented system. It relies on propagating metadata between processes and network boundaries to construct the complete journey of a request through a system, even if that journey requires communication between multiple services. A common approach to distributed tracing is to represent a flow through the system as a *trace* consisting of potentially nested *spans*, each representing some work being done in a system (see Fig. 3).

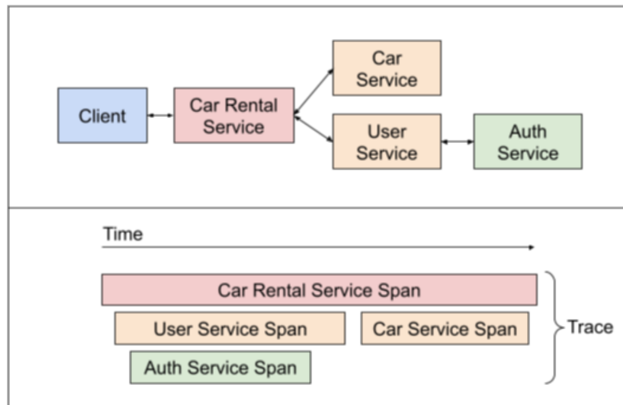


Figure 3. For a distributed system consisting of microservices, the work done on behalf of a single request can be represented in terms of a trace consisting of temporally and causally ordered spans, each representing the work being done in a service. In the above diagram, for an example Car Rental application, a request from the client to the Car Rental Service spawns a child call to the User Service, which then calls the Auth Service. Following the call to the User Service, the Car Rental Service then contacts the Car Service before returning the response to the client. The collection of spans, representing the entire request, is grouped together as a trace.

This approach to distributed tracing was introduced by Google's Dapper[21], which popularized the concepts of *traces* and spans and their use for representing the work done during a request. Following the publication of the Dapper paper, multiple distributed tracing frameworks that follow this model have been developed, including OpenTracing[17], OpenCensus[15] (Google's open-source tracing project that was spawned by Dapper), and Zipkin[26], among many others.

Distributed tracing gives visibility into the system, allowing developers the ability to see the entire path of a request through their system. It is used for debugging, system exploration, performance analysis[22], and many other purposes [25] [20].

2.2.1 OpenTelemetry.

OpenTelemetry[16] is an open source observability framework that aims to be robust, portable, and

easy to instrument across many languages. It represents a vendor-agnostic set of APIs, libraries, agents, and collector service that can be used to capture distributed traces and metrics from a microservice-oriented system. It was created by the merging of OpenTracing and OpenCensus, and it is a Cloud Native Computing Foundation[8] project.

Importantly, it represents the confluence of multiple approaches to the same philosophy of distributed tracing, and it is an attempt to establish a single open source standard that is defined, created, and led by the community. OpenTelemetry has taken the approach of defining a language-agnostic specification that "describes the cross-language requirements and expectations for all OpenTelemetry implementations"[16]. In particular, for tracing, OpenTelemetry defines the behavior for 1) the API, which is responsible for the definitions of the interfaces for trace creation, management and exporting, and 2) the SDK, which implements the definitions and interfaces in the API.

With the general specification and Protocol Buffer[24] definitions defined, OpenTelemetry leaves it up to the community to develop the language-specific implementations. The OpenTelemetry Spec reached its version 1.0 release in early 2021, and with it, API and SDK release candidates were ready for Java, Erlang, Python, Go, Node.js, and .Net[6]. The OpenTelemetry C++ API and SDK implementations reached version 1.0 later in 2021.

In addition to the specificifications above, OpenTelemetry also defines the protocol for encoding, transporting, and delivering OpenTelemetry data, aptly named the OpenTelemetry Protocol (OTLP). This specification defines the implementation of OTLP over gRPC and HTTP 1.1, including the Protocol Buffer schema for the request/response payloads, by defining a single RPC for each type of data (e.g. for the Request, it defines ExportTraceServiceRequest for traces, ExportMetricsServiceRequest for metrics, and ExportLogsServiceRequest for logs). It defines the specification for communication between two nodes in tracing architecture, though there may be more nodes (e.g. an application that talks to an agent that talks to a collector).

2.3 Envoy's Existing Tracing

Envoy allows for tracing all requests, and it also allows for flexibility with respect to the type of tracing being used. It uses a plugable tracing component, called a Tracer, that has a general API that extensions can implement for various tracing libraries. The HTTP Connection Manager uses the Tracer on request start and end, and the Tracer implementations handle the definition of what a span is, span lifecycle events (such as span creation, updating, and finishing), and exporting spans (see Fig. 4).

Envoy provides a general API for a Driver class, which defines how the tracer is instantiated, and a Span class, which defines the Span interface. Tracing extensions can implement these classes for the framework-specific requirements.

Envoy performs tracing at the application level. Once the incoming byte stream has been decoded into HTTP via the HTTP/2 Codec, the HTTP Connection Manager is responsible for the application-level lifecycle of the request. It calls the Driver's startSpan function to create a span when decoding the request's headers, and it closes the span after the response has been sent back to the client with a call to the Span's finishSpan function. This means that a Span exported by Envoy represents all of the work that Envoy does

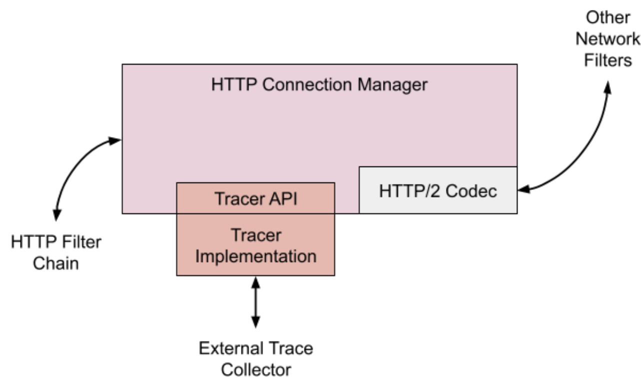


Figure 4. The structure of tracing in Envoy, with the HTTP Connection Manager using a general Tracer API to communicate with framework-specific Tracer implementations.

for the span at the application level, including HTTP filter processing, upstream connection handling, network delay for connecting to the upstream service, any work being done at or above the upstream service, and the upstream response and response handling. Importantly, for an Envoy acting as a load balancer, this means that Envoy's Span is a snapshot of how much time it took for the proxied request.

The Tracer interface also allows for Span modification. Importantly, there's a distinction between Envoy's Span class, which is a general interface that other Envoy code calls into, and the framework specific Span; for example, the OpenCensus Tracer implementation of the Envoy Span contains an OpenCensus Span as a member variable.

This general API allows framework-specific extensions to implement these interfaces, allowing for customizable tracing via pluggable extensions. For example, if one starts Envoy with the OpenCensus Tracer enabled, Envoy uses the OpenCensus implementation of the Driver. When the HTTP Connection Manager sees a new request, it calls the OpenCensus Driver's startSpan implementation, which would use the OpenCensus implementation of the Span definition. When the request is finished, the HTTPConnectionManager would finish the Span via the Span's finishSpan call. The OpenCensus implementation of the Span's finishSpan would then call an OpenCensus exporting library to export the Span.

Envoy already has Tracer implementations for OpenCensus, OpenTracing, LightStep, Zipkin, and Datadog. It also allows for external tracers that come as a third party plugin (such as Instana).

2.4 Challenges for Integrating OpenTelemetry With Envoy

The OpenTelemetry C++ library has been developed as a pluggable library to facilitate the process of adding OpenTelemetry support to microservices. This section discusses some of the challenges in creating the OpenTelemetry tracer and using the general OpenTelemetry C++ library in Envoy.

2.4.1 Threading. There is a mismatch between the threading expectations and requirements of Envoy and the OpenTelemetry C++ SDK implementation, and directly using the OpenTelemetry C++ SDK could potentially lead to performance issues in Envoy due to this mismatch.

Because each Envoy worker thread acts as a single-threaded event loop handling multiple connections, there must be as little blocking as possible. Although trace exporting is done at the end of the request's lifecycle after the response has been sent downstream and blocking won't affect the latency of the traced request, blocking could affect other requests in progress on that worker thread (See Fig. 5 for an example off-path blocking scenario). In particular, we must be careful about blocking in multiple places in the execution path: if there is a shared object that handles exporting spans, the calling code should not block while waiting for it to be free,

and that exporting object should also not have any blocking while the span exporting request is being made to the collector.

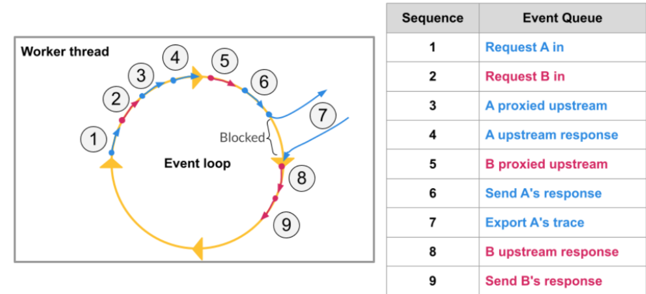
By default, the OpenTelemetry C++ SDK's OpenTelemetry gRPC exporter executes a synchronous call to the trace exporting RPC. As this could potentially introduce some blocking during the RPC call, even if it was provided with a timeout, it could introduce blocking delays into Envoy's event loop, leading to increased latency for the other requests being handled on that worker thread. To avoid this latency, a common pattern is to run the synchronous call in a background thread and use mutexes to keep a buffer of spans to export; this is what the OpenTelemetry C++ SDK does in its batch span processor[5].

The SDK implementations of the span processors also introduce potentially blocking situations, even before the exporter is called. The SDK's simple span processor relies on a custom spin lock mutex that uses lower level thread handlers like `std::this_thread::yield` and `std::this_thread::sleep_for` for gating access to the exporter's Export function. The SDK's batch span processor uses a background thread that relies on a mutex for controlling access to the exporter's Export function. Because Envoy is commonly carefully tuned to have the number of worker threads exactly match the number of cores on the host machine [10], this additional thread and synchronization usage may affect Envoy's performance.

2.4.2 Dependencies and Security.

The OpenTelemetry tracer implementation will not be able to directly use the OpenTelemetry C++ SDK due to its dependencies on libcurl.

Because of the security history of libcurl and the ongoing efforts to remove it from Envoy[23], it will not be an option for exporting the OpenTelemetry spans from Envoy. More restrictively, any dependencies for the OpenTelemetry tracer, including the OpenTelemetry C++ SDK, must not use libcurl. The main issue here is that many of the OpenTelemetry C++ SDK exporter implementations use libcurl for their HTTP calls. For example, the OTLP http exporter relies on the common http client, which uses libcurl for its HTTP requests. Reusing it for the OpenTelemetry Tracer in Envoy would further cement libcurl's position as a dependency.



Sequence	Event Queue
1	Request A in
2	Request B in
3	A proxied upstream
4	A upstream response
5	B proxied upstream
6	Send A's response
7	Export A's trace
8	B upstream response
9	Send B's response

Figure 5. Diagram of the worker thread's event loop if we block either waiting for the exporter or during the call to the exporter. In the above example, the Envoy worker thread is asynchronously handling events from two requests, A and B. The sequence is as follows: 1) Envoy receives Request A, 2) Envoy receives request B, 3) Envoy proxies request A upstream, 4) Envoy receives the upstream response for A, 5) Envoy proxies request B upstream, 6) Envoy returns A's response downstream, and 7) Envoy exports A's trace. At this point, if the request blocks, we will not be able to handle any other events while the request is in flight, and accordingly, Envoy can only handle 8) B's upstream response and 9) sending B's response downstream after the exporting call is done, even if they arrived during it. Since the worker thread is a single thread, any blocking that request A does during its life cycle has the potential to block other requests, leading to off-path caused delays.

3 Design

I designed the OpenTelemetry Tracer extension with several design axes in mind. Its threading model should mesh well with that of Envoy's; there should be no blocking and no background threads. It should have few dependencies, which

should not include libcurl. It may perform batching, but it must do so without any blocking. For creating a connection to the external trace collector, it should leverage Envoy's connection handling, rather than creating a separate RPC connection. This section discusses those design choices, the reasoning behind those choices, and alternative designs that I considered. These design axes can be found in Table 1 along with potential implementations and their tradeoffs.

3.1 Threading

I implemented the new OpenTelemetry Tracer extension using Envoy's libevent-based asynchronous gRPC library. Envoy uses this library for upstream gRPC connections, and it is implemented using the asynchronous event-based events, creating events and callbacks to be handled by the event loop. This allows for no blocking when the Tracer is exporting a span, as the RPC events are handled asynchronously like other connection events. While the export RPC is in flight, the Tracer yields control back to the worker thread's event loop, allowing other connections to be processed in the meantime.

The Tracer extension also performs no blocking when accessing shared resources. Upon initialization by the Driver, a Tracer object is instantiated in each worker thread's thread local storage, allowing access to it throughout the code running on that thread. A Tracer for each worker thread allows the threads to remain independent; since each thread has its own Tracer, there is no cross-thread bottlenecking. The Tracer contains a shared exporter class that can be accessed by each span. This exporter maintains the gRPC connection to the OpenTelemetry collector and sends RPCs on behalf of each exported span. As the OTLP allows for multiple concurrent calls, this allows for efficient sharing of the connection and exporter without the need for blocking.

Another threading design choice was the question of what to store in thread local storage to allow access across the thread level. Storing just an Exporter in thread local storage would allow for efficient RPC usage, as creating the RPC for every connection would lead to scalability issues. Going one level higher allows for shared use of system resources and access to the Exporter while allowing for future flexibility (see Section 3.5). The common Tracer can access common Envoy resources from a single location, such as Envoy's thread safe RandomGenerator, used for generating Trace and Span IDs, or Envoy's TimeSource, used for marking Span start and end times. This level of abstraction also allows for cleaner refactoring in the future if the extension is updated to use an OpenTelemetry C++ API Tracer, as it will also be desirable to share it across calls in the same thread.

The Tracer extension uses no background threads for any processing. Since the gRPC client performs the request asynchronously, there is no need for a background thread that waits for a synchronous call. Additionally, this also means the batching capability in the tracer has been implemented without the need for a background thread and locks; since the worker thread is single-threaded and the gRPC call is asynchronous, there is no need for two threads to share access to the same buffer of threads.

These approaches contrast with some of the design choices and implications in the OpenTelemetry C++ libraries. At the time of writing, the OpenTelemetry C++ SDK's gRPC exporting is synchronous by default[5]. Directly reusing it would

Potential Solutions	Design Axes			
	Threading	Dependencies	Batching	Exporting
OTLP w/ Envoy's async gRPC	Nonblocking, meshed with Envoy's worker thread libevent event loop	OTel protos	May allow batching, but without locks	Envoy's async gRPC library, leveraging Cluster Manager
OTLP w/ OTel C++ API and Envoy's async gRPC	Some blocking and assumptions about accessing global objects in the OTel API	Dependency on OTel protos and OTel C++ API	"	"
OTLP w/ OTel C++ API & SDK	Some blocking for access to global objects in the OTel API and SDK; use of background thread	Dependency on OTel protos, OTel C++ API & SDK, and third party libraries (gRPC)	May allow batching, but relies on locks	Independent RPC connection in exporter in background thread
Various output spans w/ OTel C++ API and SDK	"	Dependency on OTel protos, OTel C++ API & SDK, and third party libraries (gRPC, libcurl)	"	Various independent connection in exporter (HTTP or gRPC)

Table 1. Some of the design axes for adding the new OpenTelemetry Tracer extension to Envoy.

lead to delays in the worker thread, which is problematic. Because this call is synchronous, one approach is to use a batch exporting approach, where a background thread is created that handles the synchronous call. This is the approach taken in the OpenTelemetry C++ SDK's batch processor. While this removes the issue of explicit waiting during the RPC, it does introduce additional considerations for the cross-thread communication. The background thread reads from a buffer of spans that the main thread writes to, and this access must be coordinated with locks to prevent race conditions from occurring. If these solutions were used in the OpenTelemetry Tracer, these locks, much like the synchronous RPCs, would block the Envoy worker thread's event loop, introducing delay and latency issues.

3.2 Security and Dependencies

The only non-Envoy dependency in the new OpenTelemetry Tracer extension is the OpenTelemetry Trace Protocol Buffer definitions. Envoy already depends on the OpenTelemetry Protocol Buffers for using OpenTelemetry Logs in the OpenTelemetry logging extension, so the additional use does not add an entirely new dependency. It leverages Envoy's existing libraries such as the async gRPC client to perform the RPC connection. Envoy's job is to make requests to external backends, and it is very good at it; this new Tracer leverages that prowess.

Due to libcurl's security track record and the ongoing effort to excise it from the code base, it was not a viable dependency for the new Tracer. Additionally, since the OpenTelemetry C++ SDK relies on libcurl for parts of its span

exporting, this meant that the SDK could not be directly plugged in and used, as doing so would add libcurl as a downstream dependency. While this means the exporter does not have the built in functionality of supporting multiple exported trace types that the SDK has, Envoy already has tracer extensions for some of these types. In the future, if the SDK's dependencies change, that flexibility and direct use may be possible, but as of the time of this writing, it is not.

3.3 Batching

The OpenTelemetry Tracer allows for flexible batching after a given batching threshold. The exporter keeps a buffer of pending spans to export and checks its size against that threshold before exporting. For a batching threshold of 1, this means that Envoy would export every span immediately after the response has been returned to the client. This is the pattern followed by some other Envoy tracers as well, and for lower traffic deployments, this 1:1 pairing would allow for a more consistent cost.

For a higher batching threshold, the Tracer will wait until its pending span buffer has reached that given threshold or a given time interval has passed. It then groups the spans together and sends them as a batch to the collector. Because the worker thread executes as a single thread and the RPC call is asynchronous, this batching can be implemented without a background thread or cross-thread synchronization. The buffer will only be accessed by a single thread at a time and does not require guarding by mutex.

The tradeoff for batching extends beyond just Envoy's performance. Although Envoy may have no trouble executing an additional RPC message for each request, the upstream trace collector and the broader network may need more flexibility with regards to the volume of messages being sent. As the 1:1 span to export ratio doubles the number of requests on the network by adding an export call for every request, this batching allows for a gentler firehose of span exporting. Additionally, the OpenTelemetry may also benefit from the batching, as it is directly in the path of that firehose as well.

While the OpenTelemetry C++ SDK also allows for the flexibility of batching, it does so in a way that contrasts with Envoy's threading model (see above). In particular, it runs a background thread that is responsible for the synchronous call, then coordinates a shared buffer via mutexes. While this removes the burden of the synchronous call from the main thread, it still requires the synchronization between threads, which is potentially problematic for Envoy. While the broader system may benefit from this batching, Envoy's strict needs require the implementation to be better suited for Envoy's non-blocking needs.

3.4 Connecting to the collector

To send OTLP messages to the OpenTelemetry collector, the exporting call in the OpenTelemetry Tracer uses Envoy's asynchronous gRPC client, which is accessed by an exporter class that lives in the Tracer. This client allows the Tracer to use an asynchronous event-based call that meshes well with the worker thread's event loop. Additionally, the client ties in with Envoy's handling of other upstream requests, as it handles the connection to the OpenTelemetry Collector through Envoy's cluster management and router, similar to the way it would handle a connection to an upstream service. This allows simple configuration of the OpenTelemetry Collector as a cluster in Envoy's configuration, much like

one would configure a backend service. The connection is handled efficiently in Envoy's cluster manager system, and all parts of the connection are handled asynchronously in harmony with Envoy's event loop. This is something Envoy already does well, and the Tracer can take advantage of it for its connection needs.

One other option for the exporting would be sending the OTLP messages over HTTP. Envoy also has an asynchronous HTTP library, and one future development could be to add support for this as well. Because the Tracer accesses the gRPC client through its exporter class, it would be straightforward to add a configuration option that initializes the Tracer with an HTTP exporter or a gRPC exporter accordingly. To create that HTTP connection, it would be key for the exporter to reuse Envoy's HTTP client; as mentioned above, using libcurl would not be a viable option for it.

This approach allows the RPC connection to the collector to mesh well with Envoy's event loop. The OpenTelemetry C++ SDK and other Tracer extensions in Envoy use an independent RPC client for the connection, creating it directly from the Tracer to the collector. While this functionally works, adding the exporter as an integrated part of Envoy's upstream connection handling allows for simpler configuration and better performance.

3.5 Future extensibility and the OpenTelemetry C++ API & SDK

This implementation represents an initial effort to get OTLP support in Envoy without the overhead of adding the external OpenTelemetry C++ API and SDK dependencies.

One design decision was the balance between how much of the implementation would be custom written code and how much would be reused from the OpenTelemetry C++ API and SDK. Because of the mismatch of expectations around threading, blocking, and global variables, the initial implementation of the OTLP trace exporting does not rely on the API or the SDK. In this way, this implementation defers to the requirements of Envoy over the benefit of the shared OpenTelemetry library code.

In a way, this brings up the larger question of applying general purpose framework code to very specialized codebases. In the case of Envoy, it has a number of unique requirements, such as its threading model and its strict dependency requirements, while also containing functionality that already solves some of the general problem, like its asynchronous connection handling. Envoy is good at its job of handling connections, and leveraging that allows us to build on Envoy's strength. The OpenTelemetry C++ API and SDK are designed to be used in a wide variety of services to easily instrument them with OpenTelemetry tracing; while they are usable and applicable to many types of programs, Envoy's requirements necessitate a more specialized approach.

That being said, there is room to extend this solution in the future to use the OpenTelemetry C++ library concepts. The added Tracer class could be refactored in the future to use the OpenTelemetry C++ Tracer and TracerProvider classes. A custom implementation of the SDK could be added that implements the same behavior without locks, and it could use a custom Exporter that ties in to Envoy's async gRPC client. By keeping the Tracer abstraction between the Driver's startSpan call and the Span's export function, the implementation allows for future refactoring without major API changes.

4 Implementation

I implemented the new OpenTelemetry Tracer extension in around 1,500 lines of code. The implementation will soon be available on GitHub. To test the implementation, I wrote unit tests in C++ and confirmed the correct tracing behavior with a simple system consisting of two simple backends and the OpenTelemetry Collector (see Section 5).

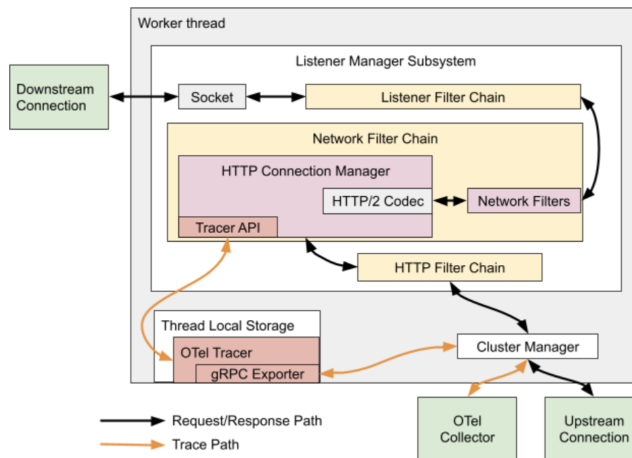


Figure 6. Diagram of the architecture of the new OpenTelemetry Tracer. The Tracer is stored in Thread Local Storage, accessible through the Tracer API from HTTP Connection Manager instances on the worker thread. Using Envoy's async gRPC client, the Exporter sends Spans to the Collector through the Cluster Manager's external connection functionality.

4.1 Implementation details

To add the new OpenTelemetry capability, I added a Tracer extension for OpenTelemetry. It consists of a factory class for instantiating the tracer (OpenTelemetryTracerFactory), a Driver class implementation, a Tracer class that contains an OpenTelemetryGrpcTraceExporter class, and an Envoy Span class implementation that contains an OpenTelemetry Span as a member variable. These were all added in a new `Envoy::Extensions::Tracers::OpenTelemetry` namespace.

The Tracer class is an intermediary parent class that contains both the exporter (the OpenTelemetryGrpcTraceExporter) used for sending the spans to the collector and methods for creating new Spans. When the Tracer creates a new Span, it passes a reference to itself to the Span constructor, which then stores it as a member variable. This reference to the parent Tracer allows the Span's finishSpan method to access the shared Exporter through the parent Tracer; since the Span finalizing call lives as a method on the Envoy Span class, this reference allows it to refer back up through the class hierarchy to access the shared Exporter. See Figure 6 for a depiction of this architecture.

Other tracers, such as the OpenCensus tracer, rely on the internal span representation referencing a static global exporter that was created in a general Get function. In the case of the OpenCensus Span, this behavior lives in framework-specific code that is used in Envoy. Since the OpenTelemetry implementation does not rely on the OpenTelemetry framework specific code, this pattern can be avoided, and the common exporter can be used without the global object.

I implemented the batching without the need for mutexes or other synchronization, as mentioned above. To allow for full flexibility, I set the batching threshold (how many spans to wait for before exporting) and batching timeout (how often to flush the spans in the buffer if the threshold is not met) to user configurable runtime configuration variables[9]. This allows Envoy users to set their own values and to modify the values without restarting Envoy. For example, if the Collector becomes overwhelmed due to the frequency of messages, the batching threshold could be dynamically set to a higher

number to relieve some of the pressure from the collector. Additionally, this lets the user define the values that are best for their deployment and gives them an additional knob to turn when tweaking the performance of their Envoys.

Additionally, since the gRPC Exporter uses Envoy's async gRPC client, it handles the connection to the OpenTelemetry Collector through Envoy's cluster management and router, similar to the way it would handle an upstream connection. This allows simple configuration of the OpenTelemetry Collector as a cluster in Envoy's configuration, much like one would configure a backend service. The connection is handled efficiently in Envoy's cluster manager system, and all parts of the connection are handled asynchronously in harmony with Envoy's event loop. For example, the following is a snippet from the Envoy configuration required to use the new Tracer, with the OpenTelemetry tracer's configuration using an upstream cluster for its RPC:

```
...
tracing:
 provider:
  name: envoy.tracers.opentelemetry
  typed_config:
   "@type": ...OpenTelemetryConfig
   grpc_service:
    envoy_grpc:
     cluster_name: opentelemetry-collector
  timeout: 0.250s
...
clusters:
 - name: opentelemetry-collector
  type: STRICT_DNS
  lb_policy: ROUND_ROBIN
  http2_protocol_options: {}
  load_assignment:
   cluster_name: opentelemetry-collector
   endpoint:
   - lb_endpoints:
    - endpoint:
     address:
      socket_address:
       address: open-telemetry
       # Default Collector OTLP gRPC port
       port_value: 4317
```

The common pattern in the OpenTelemetry C++ library is to have global objects, such as a TraceProvider that uses a global TextMapPropagator for propagating trace context and a shared SpanExporter. While this would work for most multithreaded applications, the siloed model of Envoy's worker threads requires stricter separation. The new implementation is one level lower; instead of a single shared singleton across all threads, the new Tracer is a per-thread singleton. But, even bringing the global object pattern to the thread level would clash with Envoy's thread model, as waiting for a mutex could be problematic for the single worker thread's event loop (see Section 3.1). Additionally, since this new OpenTelemetry does not use the OpenTelemetry C++ SDK (see Sections 3.1 and 3.2), the Tracer and Exporter could not just use the SDK implementation for exporting; this required writing the exporter from scratch.

4.2 Open Source process

No project exists in isolation, and this work was no exception. Working at the interface between two open source projects

offered a unique opportunity to directly engage with maintainers and experts in each project, and I am deeply indebted to the community for their help, insight, and guidance.

I initially found the project by reaching out to Harvey Tuch, one of the Envoy maintainers at Google. I began the process with an initial design proposal that I had reviewed by Envoy maintainers, OpenTelemetry contributors, engineers who had done previous work on adding tracers to Envoy, and engineers who had worked on adding OpenTelemetry logging support to Envoy. Following the initial proposal, I took on the feature request bug[1] in the Envoy GitHub issue tracker before starting development. During the process, I continued to have discussions with Envoy and OpenTelemetry maintainers; these conversations helped ensure that the direction of the project was aligned with Envoy's expectations and common OpenTelemetry practices. In particular, it was very helpful to understand the current state of the OpenTelemetry C++ libraries and confirm that the approach above was appropriate.

At the time of writing, I'm planning to open a pull request in upstream Envoy with this completed work and evaluation. This will involve some additional productionizing, code review, and shepherding through the pull request process, as well as any additional follow-up work to land this work upstream.

5 Evaluation

This section covers evaluating the new Tracer and the aforementioned design decisions.

To evaluate the design choices for this implementation, I performed benchmarking via Nighthawk, Envoy's L7 performance characterization tool[4]. Nighthawk sends a large volume of traffic to a given endpoint (in this case, pointed at an Envoy), and it has many configuration options, including the number of connections, the concurrency of the requests, and request characteristics such as request method, response body size, and desired number of requests per second (RPS). Nighthawk records latency information connection setup, request to response time, and overall total latency. Nighthawk also provides simple test server that generates predetermined responses at a configurable delay.

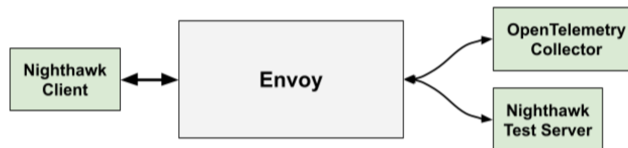


Figure 7. Diagram of the simple test system for evaluation consisting of the Nighthawk Client, Envoy, the Nighthawk Test Server as the single backend, and the OpenTelemetry Collector.

Given a duration, Nighthawk will perform as many requests as it can up to the given RPS limit for that duration. It can be run in either closed-loop mode, where it will limit its rate depending on responses from the system, or open-loop mode, where it will perform no rate limiting. To explore the effect on request latency and overall throughput, these evaluations were mostly performed in closed-loop mode.

5.0.1 Methodology. I performed these evaluations on a simple system consisting of the Nighthawk Client, an Envoy proxy, the OpenTelemetry Collector, and the Nighthawk Test Server (see Fig. 7). The Envoy running the OpenTelemetry Tracing extension used a single worker thread (with the concurrency flag set to 1) to improve comparability across runs. Sampling was manually set to always be on to ensure that every request would generate a span. The Envoy had a single backend destination service, the Nighthawk Server, which

responded with 10 bytes of the letter "a" after a static delay of 10ms (i.e. every request to the Nighthawk test server takes 10ms and returns "aaaaaaaaaa"). The OpenTelemetry Collector was added to Envoy as a static cluster. I was able to record overall latency statistics from the Nighthawk client, detailed request information from Envoy, and trace and span statistics from the OpenTelemetry collector. These evaluations were performed on a gLinux machine with 6 cores and 128 GB of RAM. Note that these evaluations are not direct measures of Envoy's capabilities as a high performance proxy; additional evaluations in a production-like environment with additional configurations would allow the best comparison (e.g. comparing Envoy to another proxy).

5.1 Comparison with Envoy without Tracing

For a confidence check for the efficiency of the new tracer, I performed benchmarking using the same Envoy binary in two different configurations, first without tracing and then with the new OpenTelemetry Tracer enabled. The OpenTelemetry Tracer used a default batching threshold of 125. For these two runs, I performed the benchmarking with Nighthawk in the open-loop mode, where it does not wait for a request to finish before sending additional requests. This allowed me to saturate Envoy and show that a similar throughput level with tracing does not lead to additional latency. In order to isolate any additional latencies, I ran the Nighthawk test server without the artificial delay, leading to the overall latencies being reflective of the time spent in the Envoys.

Each test run consisted of 60 seconds of 12 worker threads each sending a maximum of 100 calls per second with 4 connections. This resulted in 72000 total requests at 1200 requests per second being sent to the Envoy. Additionally, I was able to verify from Envoy and Collector statistics that all spans were successfully transmitted to the Collector for the run with tracing enabled. The latency results from this test can be seen in Figure 2. Note that both configurations exhibit some variation in the later latency percentiles; one could imagine that with high amounts of traffic, there would be many competing events to be handled by the event loop, and much like a random walk, some of these unlucky requests may be destined for slightly longer delays at the tail end. This comparison does confirm that this approach is in the same ballpark latency-wise without any surprising latency increases due to the new Tracer, as the latency percentiles are similar across the board and only a small difference beyond the standard deviation at the 99.9th percentile.

5.2 Batching

The next question for evaluation is as follows: how does the choice of the OpenTelemetry Tracer's batching threshold affect Envoy's performance? To evaluate this impact, I performed benchmark testing with the batching threshold set to various values ranging from 1 (i.e. no batching) to 125 (i.e. spans would be exported every 125 requests).

Because the benefit from batching would be most apparent at higher request volumes, I performed this benchmarking with a 100x limit on total traffic as the previous test, with each test run consisting of 60 seconds of 12 worker threads each sending a maximum of 1,000 calls per second with 4 connections, leading to a maximum of 12,000 RPS. Due to the saturation of the system and the closed-loop mode, the Nighthawk workers reached a lower limit of RPS; this varied across the runs as Envoy's performance changed, leading

Configuration	Mean	Std dev	Latency Percentile						
			0.5	0.75	0.8	0.9	0.95	0.99	0.999
Envoy without tracing	0.45ms	0.44ms	0.40ms	0.43ms	0.44ms	0.47ms	0.53ms	1.38ms	6.15ms
Envoy with OpenTelemetry tracing	0.48ms	0.53ms	0.42ms	0.44ms	0.44ms	0.48ms	0.59ms	1.63ms	7.39ms

Table 2. Comparing 60 seconds of 1200 RPS being sent to Envoy with and without the new OpenTelemetry Tracer.

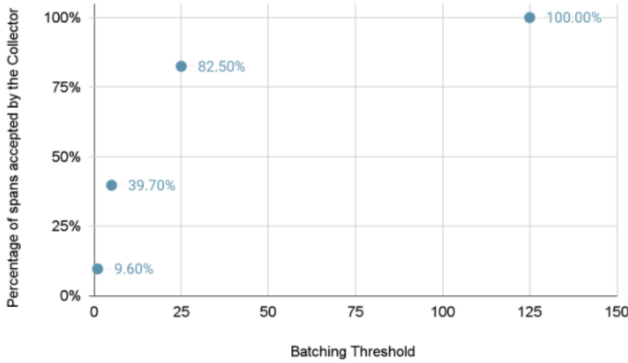


Figure 8. For a high volume evaluation, changing the batching threshold allowed the OpenTelemetry Collector to "catch up" to the spans coming from the Envoy.

to the differences in total proxied requests. This resulted in around 3,000 RPS being sent to the Envoy. The results of this evaluation can be seen in Table 3, and a visualization of the change in span acceptance can be seen in Figure 8.

The main result here is that the full collection of spans was not being received by the OpenTelemetry Collector at lower batching thresholds, although Envoy was sending a message for each exported span (as verified by the Envoy logs). This indicates that the volume of messages was too high for the Collector and that it dropped incoming spans. Additionally, the effect of the additional exporting calls can be seen in Envoy's increase in throughput (seen in the increase in number of requests Envoy was able to handle from Nighthawk) and the decrease in the 99th latency percentile as the batching threshold increased. While the exact batching values may not translate across network and system configurations (i.e. 125 is not a silver bullet batching threshold), this shows the ability of batching to help mitigate network saturation, and the dynamic configuration allows for flexible use to suit the deployment's needs. In combination with a dynamic sampling rate, this threshold will allow developers to further tune their systems.

5.3 Connecting to the collector

One of the design decisions mentioned above was the choice of using Envoy's async gRPC client instead of a separate RPC client, either in a synchronous mode (similar to the OpenTelemetry C++ SDK's simple span processor) or in a background thread (similar to the OpenTelemetry C++ SDK's batch span processor). To evaluate this design decision, I performed benchmarking with the OpenTelemetry Tracer implemented in those three ways: 1) with Envoy's async gRPC client as described above, 2) using a separate synchronous vanilla gRPC connection, and 3) using a separate synchronous vanilla gRPC connection in a background thread with batching, with the latter two not leveraging Envoy's upstream connection handling.

I implemented both additional versions with batching as well. For the synchronous gRPC in the main thread, the

Configuration	Total proxied requests	99th latency percentile	Spans received by collector	Successful requests to Collector	Percentage of total spans accepted by Collector
Envoy with OTLP tracing and no batching	172133	23.8ms	16361	16361	9.6%
Envoy with OTLP tracing and batching every 5 spans	187850	22.1ms	74630	14926	39.7%
Envoy with OTLP tracing and batching every 25 spans	191943	21.7ms	177575	7103	82.5%
Envoy with OTLP tracing and batching every 125 spans	195061	21ms	195061	1561	100%

Table 3. The performance of various batching thresholds and the downstream effect at the Collector..

Tracer keeps pending spans in a buffer calls a synchronous exporting RPC when the number of pending spans has reached the batching threshold. For the synchronous gRPC in the background thread, the worker thread appends each span to a buffer. The background thread loops continuously, and when it sees that the buffer has met the batching threshold, it exports all spans in the buffer. This solution uses mutexes to guard shared access to the span buffer to prevent race conditions. These are the approaches taken in the OpenTelemetry C++ SDK for the simple span processor, which uses a synchronous call, and the batch span processor, which executes the synchronous RPC in a background thread from a shared and guarded mutex.

Using the same system configuration as above, I evaluated the three versions with a maximum of 100 RPS from each Nighthawk worker thread for a total of 1,200 RPS to Envoy for 60 seconds. Each version used a batching threshold of 125 spans. After each run, I was able to verify from the Collector logs that no spans were dropped. The results are in Table 4.

As expected, the synchronous gRPC performance was much worse, resulting in a 26ms increase in the mean latency and an increase of nearly 55ms in the 99th percentile latency. Since the Nighthawk Test Server introduces an artificial delay of 10ms, meaning that the Envoy-specific delay in the mean time increased from 1.6ms to 26ms. This is further evidence that we do not want to block the event loop. While the use of the background thread allowed the third version to have better performance with the synchronous RPC running in the background, it still showed signs of worsened performance at the higher percentiles. While the majority of the requests had similar performance to the use of Envoy's asynchronous client, these results suggest that there is a tail segment of requests that do get blocked while waiting on the

shared mutex that coordinates between the worker thread and the background thread.

I next performed an evaluation with an increase in volume of requests to the Envoy. Much like the firehose test from before, each worker thread was allowed a maximum of 1000RPS, resulting in a maximum of 12,000 RPS to Envoy, though like the earlier evaluation, resource contention resulted in fewer RPS than the maximum (around 3,200). The results are shown in Table 5. While the mean and median latencies are similar, the implementations diverge after the 90th percentile, with the tail latencies in the background thread implementation nearly 14ms longer than those of the async implementation. This corresponds to more delays due to blocking and synchronous requests; as the background thread accesses the shared buffer, the worker thread must wait before appending spans to the buffer and the RPC call is limited to sequential calls. This may be improved by increasing the batching threshold further, but there is an additional tradeoff between a larger RPC message size and the duration of the RPC, as well as the volume of incoming spans that the Collector can handle. This could also be improved through more advanced lockless programming techniques, though the tested design is that of the C++ SDK and allows us to directly compare the async gRPC with it.

5.4 Comparison with an existing Tracer extension

As an additional confidence test, I performed an evaluation comparing the new OpenTelemetry Tracer with the behavior of the Zipkin Tracer. The Zipkin tracer has a similar design, where it maintains a buffer of spans and exports them at a given batching threshold. The Zipkin performs its exporting via Envoy's async HTTP client; much like the async gRPC client, it performs its exporting via Envoy's built-in connection handling. I performed 60 seconds of 100 RPS from each Nighthawk thread, leading to 1,200 RPS to Envoy, and I was able to verify via logs that all 7200 spans were accepted by the OpenTelemetry Collector. Note that because the OpenTelemetry Collector also accepts Zipkin traces, I was able to reuse it and verify the span receipt in a similar way to the OpenTelemetry spans. The results from this evaluation are in Table 6.

While not a perfect comparison between these configurations, this does confirm that this approach is in the same ballpark latency-wise without any surprising latency increases due to the new Tracer.

6 Future Work

This is the first step at adding OpenTelemetry tracing support to Envoy. Although it represents a subset of the possible features of the OpenTelemetry C++ SDK, it does represent the meshing of OpenTelemetry tracing and Envoy's strict requirements. Future work could include the refactoring of this solution to include the headers and structure of the OpenTelemetry C++ API and SDK, though the implementations would need to align with Envoy's requirements and goals. This could be done in parallel with hardening and improvements to the OpenTelemetry C++ client code; for instance, the gRPC exporter could take an existing stream or client instead of creating its own. That being said, this still arrives at the same question as earlier: how specialized should general-use framework code be made in order to fit a very specialized application?

Additionally, as the OpenTelemetry C++ library continues to develop and harden, an interesting future workstream

Configuration	Mean	Std dev	Latency Percentile						
			0.5	0.75	0.8	0.9	0.95	0.99	0.999
Envoy's asynchronous gRPC client	11.6ms	1.2ms	11.0ms	12.6ms	12.7ms	12.7ms	12.8ms	13.8ms	19.2ms
Synchronous gRPC	36.0ms	8.1ms	36.0ms	40.2ms	41.2ms	45.5ms	51.6ms	58.7ms	63.9ms
Synchronous gRPC in background thread	11.6ms	1.5ms	11.5ms	12.6ms	12.7ms	13.2ms	13.7ms	16.2ms	22.9ms

Table 4. Latencies when comparing 60 seconds of 120 RPS being sent to Envoy with the OpenTelemetry Tracer implemented in three separate ways: with Envoy's asynchronous gRPC client, with a synchronous gRPC client, and with a synchronous gRPC client running in a background thread.

Configuration	Mean	Std dev	Latency Percentile						
			0.5	0.75	0.8	0.9	0.95	0.99	0.999
Envoy's async gRPC w/ batching=125	14.8ms	2.3ms	14.7ms	16.3ms	16.6ms	17.8ms	18.9ms	21.0ms	23.2ms
Synchronous gRPC in background thread w/ batching=125	14.7ms	3.0ms	14.3ms	16.3ms	16.7ms	18.2ms	19.4ms	24.7ms	37.4ms

Table 5. Latencies from the comparison of batching with Envoy's async gRPC client and batching with the synchronous call in a background thread.

Configuration	Mean	Std dev	Latency Percentile						
			0.5	0.75	0.8	0.9	0.95	0.99	0.999
Envoy with Zipkin tracing	11.5ms	1.4ms	11.0ms	12.6ms	12.9ms	13.0ms	13.4ms	14.8ms	19.7ms
Envoy with OpenTelemetry tracing	11.6ms	1.2ms	11.0ms	12.6ms	12.7ms	12.7ms	12.8ms	13.8ms	19.2ms

Table 6. Latencies from comparing the new Envoy OpenTelemetry Tracer with the Zipkin tracer, which shares some of the design choices.

may be allowing for multiple different types of traces. While Envoy currently has a separate tracer for each framework, Envoy itself could represent the bridge between Envoy's spans and the external frameworks. The OpenTelemetry C++ allows for the use of plugable exporters to export framework-specific spans, and the OpenTelemetry Tracer extension could leverage that ability and be the single point of tracing for Envoy.

The intersection of Envoy and distributed tracing is still very open for future work, including tracing at just the proxy level (what would a system's trace's look like when only the

proxies are instrumented), control plane work (how could Envoy's dynamic configuration be augmented with tracing information like Jaeger's dynamic sampling configuration [3]), more dynamic asynchronous control over tracing via multiple queues of varying levels of priority[7], and distributed-tracing based performance analysis[22] of microservice architectures via proxy-level tracing.

7 Conclusion

Distributed tracing is the practice of bringing observability to a microservice-oriented system. It relies on propagating metadata between processes and network boundaries to construct the complete journey of a request through a system, even if that journey requires communication between multiple services. OpenTelemetry, an open-source standard and framework for distributed tracing, has emerged as the front runner standard in distributed tracing in industry. Envoy is a high performance C++ distributed proxy that is commonly used in modern service-mesh architectures. This new OpenTelemetry tracing support to Envoys allows for the efficient exporting of OTLP traces from Envoy, and it sets the stage for future development with the OpenTelemetry framework.

Acknowledgments

I would like to thank Raja Sambasivan, Fahad Dogar, and Mark Hempstead for serving as my Thesis defense committee, and I would particularly like to thank Raja for all of his help as my advisor and mentor for this project, from getting me interested in distributed tracing during his Cloud Computing class to helping me get this project to the finish line. I am also in deep gratitude to my DOCC lab colleagues, who helped me refine the project, writing, and explanations, even though they had to hear me present it at least a half dozen times.

I would also like to thank Harvey Tuch and my other colleagues at Google that have helped me and given me guidance, including everything from the initial idea and early design reviews to feedback on my lightning talks and PRs. Having a deep bench of technical knowledge on Envoy and OpenTelemetry helped a great deal, and I very much appreciate all of the assistance.

Finally, I would like to thank my wife Elizabeth for her unwavering support during my time at Tufts, and I must also thank my dog Karl, who helped to get me outside and thinking on long walks.

References

[1] [n.d.]. *tracing: Transition to OpenTelemetry from OpenTracing and OpenCensus.* https://github.com/envoyproxy/envoy/issues/9958

[2] 2021. Istio. https://istio.io/.

[3] 2021. Jaeger Sampling. https://www.jaegertracing.io/docs/1.28/sampling/.

[4] 2021. Nighthawk: A L7 (HTTP/HTTPS/HTTP2) performance characterization tool. https://github.com/envoyproxy/nighthawk.

[5] 2021. OpenTelemetry C++. https://github.com/open-telemetry/opentelemetry-cpp.

[6] 2021. OpenTelemetry specification v1.0 enables standardized tracing | Google Cloud Blog. https://cloud.google.com/blog/products/operations/opentelemetry-specification-enables-standardized-tracing

[7] Hafiz Mohsin Bashir, Abdullah Bin Faisal, M Asim Jamshed, Peter Vondras, Ali Musa Iftikhar, Ihsan Ayyub Qazi, and Fahad R. Dogar. 2019. Reducing Tail Latency Using Duplication: A Multi-Layered Approach. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 246–259. https://doi.org/10.1145/3359989.3365432

[8] Cloud Native Computing Foundation. 2021. https://www.cncf.io/.

[9] Envoy Project Authors. 2021. Runtime configuration. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/operations/runtime.

[10] Envoy Project Authors. 2021. What are best practices for benchmarking Envoy? https://www.envoyproxy.io/docs/envoy/latest/faq/performance/how_to_benchmark_envoy.

[11] Envoy Proxy. 2021. https://www.envoyproxy.io/.

[12] libevent. 2021. https://libevent.org/.

[13] Matt Klein. 2017. Envoy threading model. https://blog.envoyproxy.io/envoy-threading-model-a8d44b922310.

[14] NGINX. 2021. https://www.nginx.com/.

[15] OpenCensus. 2021. https://opencensus.io/.

[16] OpenTelemetry. 2021. https://opentelemetry.io/.

[17] OpenTracing. 2021. https://opentracing.io/.

[18] John Ousterhout. 1996. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, Vol. 5. San Diego, CA, USA.

[19] Michael Payne and Harvey Tuch. [n.d.]. *Understanding, maintaining and securing Envoy's supply chain.* https://www.youtube.com/watch?v=0VRY1FkeKxw

[20] Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. 2014. So, you want to trace your distributed system. *Key design insights from years of practical experience. Parallel Data Lab* (2014).

[21] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[22] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K Coskun, and Raja R Sambasivan. 2021. Automating instrumentation choices for performance problems in distributed applications with VAIF. In *Proceedings of the ACM Symposium on Cloud Computing*. 61–75.

[23] Harvey Tuch. [n.d.]. *Remove curl as an Envoy dependency.* https://github.com/envoyproxy/envoy/issues/11816

[24] Kenton Varda. 2008. *Protocol Buffers: Google's Data Interchange Format.* Technical Report. Google. http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html

[25] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing performance problems with temporal provenance. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 395–420.

[26] Zipkin. 2021. https://zipkin.io/.