Pixie's protocol tracing with eBPF

Omid Azizi, Yaxiong Zhao, Ryan Cheng, John P. Stevenson, Zain Asgar
{oazizi, yzhao, rcheng, jps, zasgar}@pixielabs.ai
Pixie Labs

Introduction

When debugging modern microservices applications, developers frequently turn to traces, metrics and logs to root-cause issues. While powerful, these data sources often require that the application have been manually instrumented. Unfortunately, instrumentation takes a lot of effort, resulting in software systems that are often inconsistently instrumented.

Pixie [1, 2], a CNCF sandbox open-source project, aims to close these observability gaps by providing automatic observability using eBPF [3]. Pixie's approach to observability has the following characteristics:

- No-instrumentation: Pixie's approach is to gather metrics, logs and events without requiring any code changes or even application redeployments. Much of this functionality is enabled by eBPF.
- Data-oriented UI: Pixie enables users to query data through a Pandas-inspired [4] scripting language, and view them through a Web UI.
- Kubernetes-native: Although the core features are general to most Linux systems, Pixie is specifically designed to provide instant monitoring of applications in Kubernetes environments.

In this paper, we present Pixie's protocol tracer, which provides automatic message tracing. With the growing adoption of Kubernetes and complex microservices-based applications, understanding the messages between the various services becomes critical in managing Kubernetes applications. The protocol tracer's goal is to provide instant observability to developers to understand and monitor the workings of their Kubernetes applications—all without requiring any manual instrumentation.

Background

Observability in general, and distributed message tracing [5] in particular, is well-known in the developer community. Instrumentation-based approaches, like Jaeger [6], OpenTelemetry [7], Zipkin [8], provide standard APIs and libraries for application developers to add tracing functionality for distributed systems. Manual instrumentation is required, but the inclusion of trace IDs enables one to follow requests through the system.

Network tracers like tcpdump [9] and Wireshark [10] capture all network traffic directly from inside the network stack using libpcap. They exemplify the same no-instrumentation capability as Pixie's protocol tracer. The main difference is that these network tracers operate at the network layer (and thus captures TCP packets), while Pixie operates closer to the application layer. Nevertheless, both can capture and parse protocol messages.

The Pixie project leverages eBPF via the BCC [11] and bpftrace [12] projects. The protocol tracer in particular uses BCC to manage its eBPF code. BCC was selected because of our need to interface with our eBPF probes in an optimized manner, and our need to write highly customized eBPF code.

Architecture

Figure 3 shows the high-level approach to the Pixie Protocol Tracer: eBPF *kprobes* are deployed on 17 networking-related syscalls. These probes copy the application's messaging data to user space via perf buffers.

To avoid sending too much irrelevant data, some preliminary filtering is applied in the eBPF code; more filtering is then performed in user-space. Some examples of traffic that we exclude in BPF are:

- Address families that are not of interest (e.g. Unix domain sockets).
- Traffic for protocols that are not of interest, using a simple rule-based traffic classifier.
- `read()` and `write()` calls to files rather than sockets.

Data and meta-data that is sent to user space is organized into *connection trackers*. A connection tracker manages the entire lifecycle of a network connection. Its functionalities include:

- Tracking connection metadata from the BPF runtime, and resolving the remote endpoint if not traced.
- Parsing data into structured protocol messages for the appropriate protocol.
- Exporting records to Pixie's distributed time-series database.

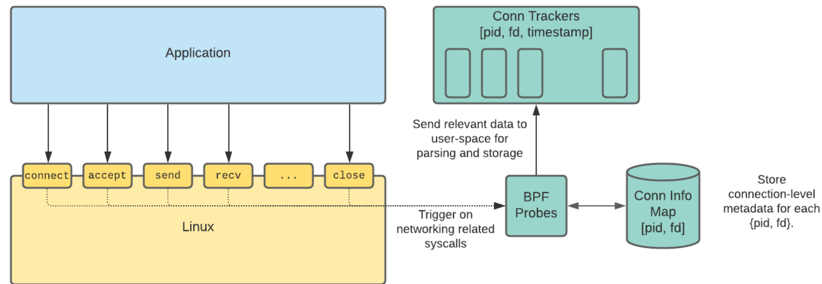

Figure 3. Overview of the Pixie Protocol Tracer. Networking-related syscalls are traced with BPF probes. Relevant Data and Meta-data is sent to user-space for further processing into structured events.

Syscall variants

One of the main challenges of tracing syscalls is the multitude of ways Linux syscalls can be used for networking; there are various syscalls that may be used for the same purpose, and some syscalls can be used in different ways. Reading data from varying data structures from these syscalls is also challenging.

The Pixie protocol tracer deploys kprobes for the following syscalls and kernel functions:

Connection syscalls	**Recv syscall variants**	**Write syscall variants**	**Special purpose probes**
connect, accept, accept4, close	read, readv, recv, recvfrom, recvmsg, recvmmsg	write, writev, send, sendto, sendmsg, sendmmsg, sendfile	sock_alloc, sock_sendmsg, sock_recvmsg

In general, for each of these syscalls, we have two kprobes: an entry probe and a return probe. Function arguments are stored in a BPF hash map in the entry probe, and are retrieved in the return probe. The arguments and the return code are used to generate an event into a perf buffer.

Several syscalls have specific characteristics that make them difficult to trace:

Syscall	**Description**
accept, accept4	This syscall populates the remote endpoint when the caller provides the appropriate struct. However, when the caller leaves the argument as NULL, we cannot trace the remote endpoint. To work around this, we have also traced the sock_alloc symbol from which we get the missing information. We also have user-space code to resolve endpoints in case the initial connection was not traced.
recvmsg, sendmsg, recvmmsg, sendmmsg	These syscalls perform scatter-gather style send and recv. We have loops to iterate over the various buffers in these cases, but our loops are bounded due to the eBPF instruction limit on older kernels. In cases where the number of buffers is greater than our supported amount, we will lose data.
sendfile	Sendfile is an optimized syscall for efficient transfer of file data. Unfortunately, this means we cannot directly trace the data using the syscall kprobe. We are still investigating alternative ways of tracing this data.
read, write	These syscalls are used for various purposes, notably file IO. We include special probes on sock_sendmsg and sock_recvmsg to detect when these syscalls are used for networking so that we can include the appropriate.

Protocol inference in BPF

Filtering is used to avoid transferring unnecessary data through perf buffers. The primary method of filtering is protocol inference, which attributes network connections to known protocols, and only transfers data that belongs to supported protocols. As of the writing of this paper, we perform protocol inference in BPF with a series of simple rules. For example, for detecting HTTP, we expect to see strings like “GET” or “POST” at the beginning of the payload.

This approach was sufficient with our initial implementation; we have, however, expanded the number of protocols we trace. We currently support 9 protocols, and more are expected. Moreover, many of the supported protocols are binary protocols.

There are two challenges with increasing the number of protocols. First, as the list grows, we keep increasing the number of BPF instructions in use; this is a problem when deploying on old kernels that have the 4096 instruction count limit. Second, with the increased number of binary protocols, we have run into cases where protocols are mis-classified.

To solve these problems, we are now considering moving the protocol inference to the user-space code. In this approach, we seek to send a sample of the data for each connection, rather than all the data to avoid significant performance penalties.

Tracing gRPC

The HTTP/2 protocol is the next generation of the HTTP protocol, and serves as the foundation of other protocols like gRPC. A prominent feature of the HTTP/2 protocol is its HPACK [13] header compression algorithm, which includes a dynamic dictionary mapping previously observed header values to unique numeric codes. Without knowing the dictionary, we cannot decode the HTTP/2 headers. At the same time, the headers contain some of the most important information on a connection, including the `:path` field of a request.

We decided that replicating the HPACK algorithm would not meet our goals, because (1) we may have to trace connections that were made before we began tracing, and (2) we need to be robust to lost data transfers from eBPF to user-space through the perf buffers.

Our current solution uses uprobes to directly trace the gRPC library itself. As of the writing of this paper, we have applied this approach to Golang’s gRPC library. In our implementation for the Golang gRPC library, we use 5 uprobes to collect the data that we need:

Uprobe	**Purpose**
google.golang.org/grpc/internal/transport.(*loopyWriter).writeHeader	Trace headers sent
google.golang.org/grpc/internal/transport.(*http2Server).operateHeaders google.golang.org/grpc/internal/transport.(*http2Client).operateHeaders	Trace headers received
golang.org/x/net/http2.(*Framer).WriteDataPadded	Trace data sent
golang.org/x/net/http2.(*Framer).checkFrameOrder	Trace data received

At each of these trace points, we directly collect information like the header values and the payload data. We also use DWARF symbols to find the memory layout of the runtime data structures, in order to obtain the file descriptor of the connection so that we can assemble the traced data into the appropriate connection trackers.

SSL/TLS Tracing

Tracing of SSL/TLS connections is another main use case for the Pixie platform. In this case, our kprobe-based sees only encrypted traffic and we’re not able to provide the user with any observability. One option here would be to request keys from the user and to decrypt the channel. However, in the spirit of our no-hassle approach, we were looking for a solution that would work without requiring any user involvement.

We directly trace the API between the application and the SSL/TLS library to trace the traffic, as shown in Figure 4. This approach has worked well because of the symmetry between TLS library interfaces and the Linux syscalls for send and receive.

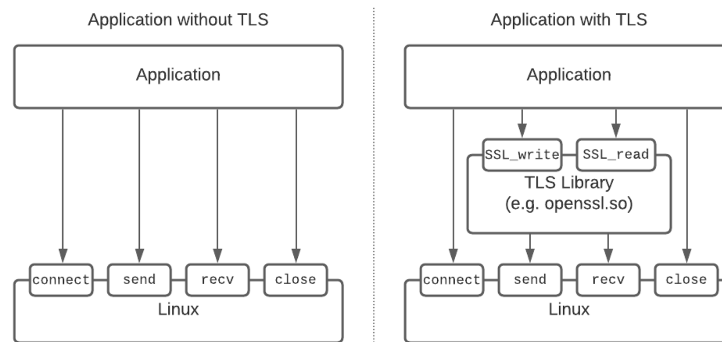

Figure 4: Comparing an application with and without TLS. The applications send and receive calls effectively go through an extra encryption layer.

We have implemented this approach for two different TLS libraries: OpenSSL and the Golang `crypto/tls` library. Our implementation contains the following uprobes:

Non-encrypted traffic kprobes	OpenSSL uprobes	Golang crypto library uprobes
send/write/etc.	SSL_write	crypto/tls.(*Conn).Write
recv/read/etc.	SSL_read	crypto/tls.(*Conn).Read

Since there is a one-to-one mapping between the read and write functions of the SSL/TLS libraries to the recv and send functions in our syscall tracing, we simply send data through the same perf buffers that are used to report non-encrypted traffic. This meant that we required no user-space changes to other components of Pixie.

Conclusion & Future Work

The open source Pixie project's goal is to produce hassle-free observability tools for the community. Pixie's eBPF-based protocol tracer, in particular, probes Linux kernel syscalls and application library functions to implement automatic protocol tracing. Combined with the UI and query-engine of the entire Pixie platform, it provides instant observability into the messages of microservice-based Kubernetes applications.

In the future, we are considering several major improvements: We are working to expand our protocol coverage to include additional popular protocols. We also plan to expand our coverage of gRPC tracing to languages like C++, Rust and Java. We are also working to simplify the protocol parsing framework to make it easier for open source contributors to implement tracing for new protocols as pluggable modules.

References

1. https://px.dev
2. https://github.com/pixie-io/pixie
3. https://ebpf.io/
4. https://pandas.pydata.org/
5. B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure,” Google, Inc., Tech. Rep., 2010. https://research.google/pubs/pub36356/
6. https://www.jaegertracing.io/
7. https://opentracing.io/
8. https://zipkin.io/
9. https://www.tcpdump.org/
10. https://www.wireshark.org/
11. https://github.com/iovisor/bcc
12. https://github.com/iovisor/bpftrace
13. HPACK: Header Compression for HTTP/2 https://datatracker.ietf.org/doc/html/rfc7541