Optimizing Envoy's Network Stack with Kmap

Joshua Levin
Brown University

Peter Cho
Brown University

1 Introduction

The development of the microservice and servicemesh has brought the sidecar container to the forefront of distributed services. These sidecares provide shared features among services deployed on the servicemesh. However, these sidecars are often interjected in a service-agnostic way by modifying iptables to redirect network communication through the sidecars. This process can introduce significant latency to responses. We focus on the Envoy [1] sidecar used by the Istio [4] servicemesh.

We present Kmap as an alternative library loaded via LD_PRELOAD using shared memory to optimize data communication between processes. Our work focuses on two key aspects:

1. Building an efficient data path between Envoy and services
2. Developing a robust API which mirrors the original network stack

Our initial prototype explores challenges and design paths for the development of Kmap. We highlight challenges for implementing Kmap for production network systems. Then, we present our prototype working on sample C/C++ webservers.

2 Background

Recent shifts to microservice architectures and the resulting servicemesh have prioritized development and deployment efficency over performance. Microservices deployed on a servicemesh (i.e. Istio [4]) use proxies which are co-located with microservices. These proxies intercept and control networking communication between services. The proxies are relatively lightweight, but use the networking stack for communication with the co-located service (Figure-2). They do this to preserve a common interface for services– the networking stack. Istio uses Envoy [1] which is the tool we focus on here for Kmap.

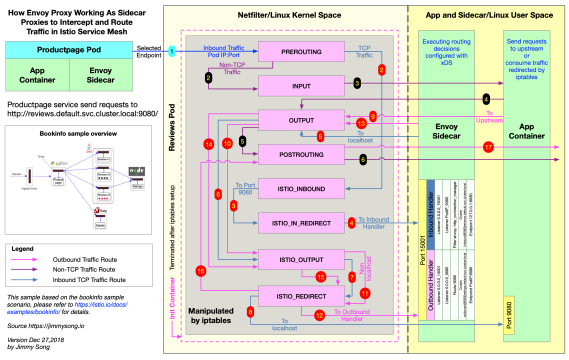

Figure 1: Envoy Network Stack [6]

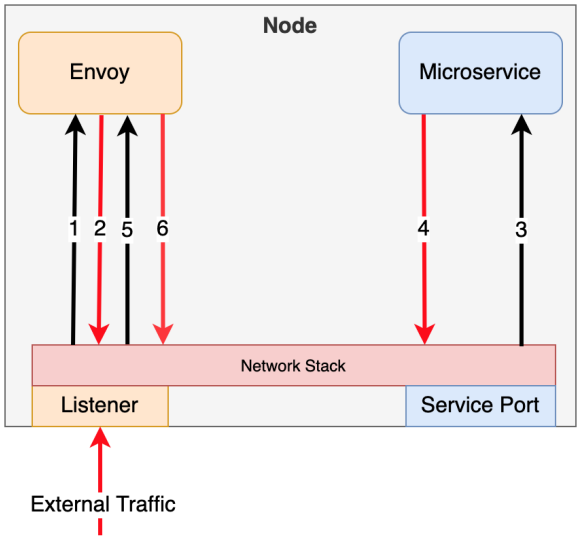

Figure 2: No Kmap Envoy Network Stack

3 Kmap

Our design for Kmap focus on avoiding costly, unnecessary invocations of the network stack. Envoy prides itself on being

application agnostic, which leads to the helpful ability to auto-inject Envoy into microservices. However, this abstract requires unnecessary invocations of the network stack. In Figure-2 we highlight in red all the calls to the network stack for the path of a single request. Then, in Figure-3 we show the reduction in those calls by using Kmap instead of the network stack for local data transfer.

Kmap works by using LD_PRELOAD to load augmented network calls before libc regular calls. Then, when Envoy and the microservice invoke read or write calls locally, rather than passing the data into the network stack, we use Kmap's shared buffers to efficiently transfer the data. Thus, we must load the shared library for both the Envoy sidecar and the microservices. The two critical challenges to realizing Kmap are:

1. Building a robust, efficient shared buffer *faster* than the network stack

2. Determining in a microservice-agnostic way which socket calls should use Kmap

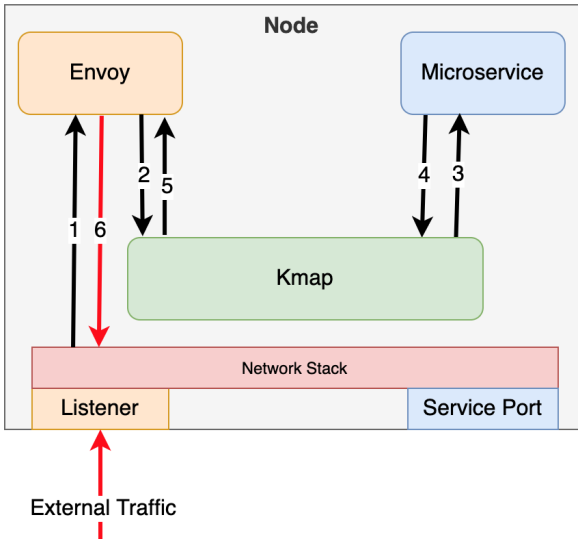

Figure 3: Kmap

3.1 LD Preload

The Linux LD_PRELOAD command [5] allows users to preload particular libraries ahead of the conventional linker and loader. This enables users to load symbols before classic libraries (e.g. libc, syscalls). Commonly this is referred to as the "LD_PRELOAD trick". For Kmap, we use LD_PRELOAD to load the Kmap libraries before conventional libraries such as libc. Inside each Kmap library we use *dlsym* to load the "real" libc functions we wrap.

3.2 Pipe options

Here, we outline potential methods for implementing the buffer Kmap uses to pass data between Envoy and the microservice.

Named Pipes: Named pipes, (FIFO) are blocking, unidirectional I/O buffers for passing data between two processes. The pipe must be opened for both reading and writing before being written two. Named pipes are traditionally slow, only offering slight speed advantages over TCP [3]. Further, they are un-directional which makes them less accessible or interchangeable compared to TCP.

Unnamed Pipes: Unnamed pipes are slightly faster than Named pipes, but are created per-process. They traditionally are used when a process forks, as they both will share a reference to the pipe. This makes them particularly tricky to implement across two independent processes and thus unhelpful for Kmap.

Shared Memory: Shared memory is a robust API which allows processes to share use of a memory region (*schm_open*). This requires mapping the same underlying memory region into the virtual memory of each process (*mmap*). Shared memory is concretely faster than pipes and the primary API Kmap uses for communicating information. Shared memory has been benchmarked to be 170 times faster than TCP sockets for communicating information between processes [3]. However, shared memory does not directly provide a buffer interface like TCP, ans so Kmap must implement that as part of its library.

3.3 When to apply Kmap

The network stack has a very well defined, robust API which applications use to communicate. A particular challenge for Kmap is pre-loading in front of those network calls and knowing when to pass through the call, or when to route to the local buffer. Since Kmap is designing primarily for Envoy, we use information about how it communicates with microservices to determine which file descriptors should use Kmap. This approach is not directly applicable to other sidecars (i.e. Linkerd) but is generalizable for applications, provided the applications work with Envoy first.

4 Prototype

Here, we discuss our current prototype, challenges, and target applications. Currently we compile a single shared object ({envoy,service}.so) for each side using compiler flags. It is imperative we intercept syscalls on each side, knowing which side of communication the library is running on to determine which network calls to intercept. Our library is written in C and is approximately 400 lines of code.

4.1 Kmap Buffer

Our implementation of Kmap buffer follows the producer/consumer model. We map a shared structure which includes directional buffers. We synchronize access using semaphores between the two processes. This is critical as both applications need blocking and non-blocking, as well as signal-based calls to determine when new data has arrived.

Function	Envoy	Service
socket	X	X
connect	X	
listen		X
accept		X
poll		X
select		X
send		X
sendto		X
sendfile		X
write	X	X
read	X	X
writev	X	
readv	X	

Table 1: Library Functions Linked (HTTP/Flask only)

Function	Envoy	Service
socket	X	X
connect	X	
listen		X
accept		X
write	X	X
read	X	X
writev	X	
readv	X	

Table 2: Library Functions Linked (Tiny C Webserver only)

4.2 Kmap Network Calls

The most challenging of Kmap's design is properly intercepting and replicating the behavior of network system calls. This aspect is especially challenging because Envoy uses unique handlers for request types (UDP, TCP, HTTP{1,2,3}, gRPC, Quic). Further, the service itself may use the network stack in obtuse ways. Our investigation of Flask [2] has revealed a number of idiosyncrasies with how Flask uses the network stack. To make Kmap generalizable, you would truly have to intercept very single network call (or possible network related). This task is enourmous. For our prototype we have focused on HTTP. In Table-1, we show the calls on each side we link to use Kmap for flask. Currently our prototype works with our tiny C server (Table-2).

4.3 Initial Results

Blah

References

[1] Envoy proxy - home. https://www.envoyproxy.io/.

[2] Flask. https://flask.palletsprojects.com/en/1.1.x/.

[3] Ipc performance. https://stackoverflow.com/questions/1235958/ipc-performance-named-pipe-vs-socket/54164058#54164058.

[4] Istio. https://istio.io/.

[5] Ld_preload documentation. http://man7.org/linux/man-pages/man8/ld.so.8.html.

[6] SONG, J. https://jimmysong.io/.