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. Istio has benchmarked Envoy to add between 5-10ms to response latencies on average for small to medium sized meshes (<=16 concurrent connections) [1]. We focus on the Envoy [3] sidecar used by the Istio [6] 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 classic POSIX sockets, which most common languages and tools use for interacting with the kernel stacks

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 [6]) 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-3). They do this to preserve a common interface for services– the networking stack. Istio uses Envoy [3] which is the tool we focus on here for Kmap.

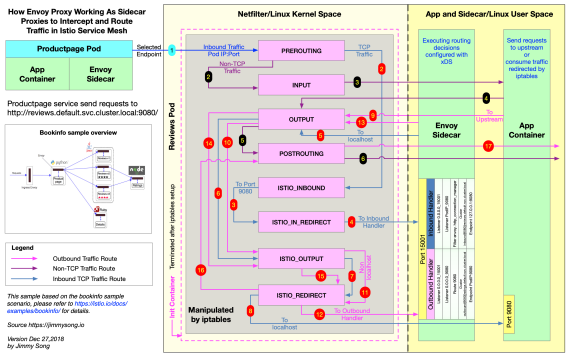

Figure 1: Envoy Network Stack [10]

3 Kmap

Our design for Kmap focus on avoiding costly, unnecessary invocations of the network stack. These network calls involve copying of bytes between two processes and kernel context switches. Avoiding the extraneous copying and reducing the frequency of context switches should greatly improve data transfer speed. Envoy prides itself on being application agnostic, which leads to the helpful ability to auto-inject Envoy into microservices. However, this abstraction results in unnecessary invocations of the network stack. In Figure-3 we highlight in red all the calls to the network stack for the path of a single request. Then, in Figure-2 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 Kmap before libc regular calls. Kmap contains several functions which match the signature of libc network calls and thus when applications invoke those calls, Kmap's version of those functions

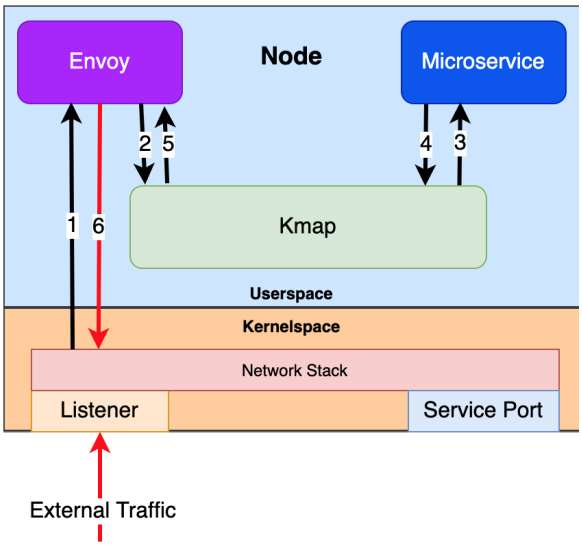

Figure 2: Kmap

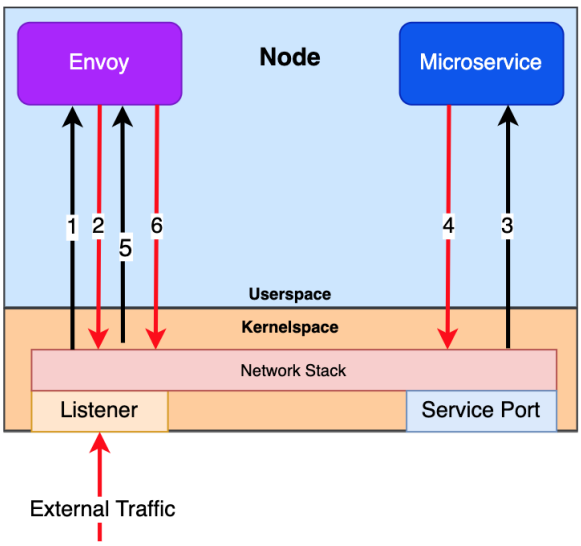

Figure 3: No Kmap Envoy Network Stack

is invoked instead of libc. Using the Kmap functions, Envoy and the microservice call read and write passing data through userspace rather than passing the data into the network stack, using 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 system that's *faster* than the network stack

2. Determining in a microservice-agnostic way which POSIX network calls should be preloaded by Kmap

3.1 Intercepting Network Calls

Direct libc Modification: One method of implementing Kmap would be to modify libc in environment for Envoy and the service. This option, however, would require extensive infrastructure modification to ensuring Envoy and the service each boot with the proper libc version (compiled for them). Further, the library change would effect any other process running the in the same environment (container or system) and thus Kmap would have to handle calls from neither Envoy or the service.

LD_PRELOAD: The Linux LD_PRELOAD command [7] 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" [2]. The LD_PRELOAD command modifies the linking process of the particular process it is attached to, requiring no modification of the default library and not interfering with any other process.

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. Thus, we can use libc functions in Kmap, passing through most calls, while intercepting and adjusting only the subset relevant to Kmap. From a maintainability perspective, this allows libc and Kmap to evolve independently while remaining compatible.

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 [5]. 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 [5]. 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 Dual Pipes

Since Kmap mirrors the network stack for two unique clients, Envoy and the service, we must implement two circular buffers. These buffers are unidirectional, providing read/write direction for each service. A core function of the POSIX network stack is blocking write and reads using syscalls like *select* and *epoll*. As such, we use have dedicated synchronization for each buffer using pthread mutexes and conditional variables. We display these aspects of Kmap in Figure-4

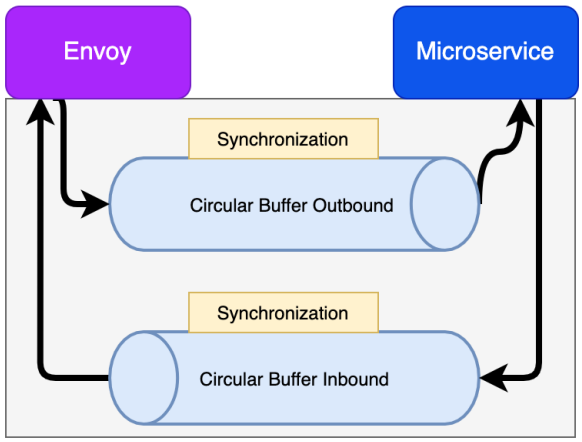

Figure 4: Kmap Design

3.4 When to apply Kmap

The network stack has a very well defined, robust API commonly called POSIX sockets. A particular challenge for Kmap is pre-loading in front of these 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. The Envoy codebase uses unique handlers for different requests (e.g. gRPC, TCP, HTTP 2, HTTP 3, QUIC) and as such Kmap will have to adjust it's structure to match the protocol in use. For now the focus of our Kmap prototype, discussed in Section-4 is Envoy's HTTP stack. We do not view Kmap as generalizable to other Envoy paths or other tools (e.g. Linkerd) without thorough inspections of the pathways. It is likely for Kmap to generalize it would require the tool and path as part of its compilation process.

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, and as such we must know 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 800 LOC.

4.1 Kmap Buffer

The implementation started with named pipe as the main Inter-Process Communication (IPC) mechanism. However, due to its unidirectional nature and difficulty in incorporating synchronization mechanisms, we decided the named pipes will not suffice.

As a solution, we used `shm_open(2)` to open a file descriptor associated with a region of shared memory and map it to the process using the `mmap` syscall. We allocate a circular buffer of size 2^{24} bytes in this shared region and read and write from it directly using `memcpy`. We will discuss our choice of the buffer size and how it affects the performance in later section. We allocate two such buffers to provide communication in each direction.

4.2 Concurrency Challenges

The Envoy process will write the outer request to the buffer and read the service response from the buffer; the service process will read the request (passed on by Envoy) and write the response, as returned by the service. We now have a classic producer-consumer problem.

Our first attempt at the solution was semaphore. We allocated two sempahores, each associated with one buffer, for the purpose of waiting and notifying. However, semaphores did not provide a convenient way to guarantee exclusive access to the buffer. This allowed for data races and caused the program to malfunction.

As an improvement, we moved to using mutexes across processes. Each buffer has a mutex associated with it and also a condition variable. We use these two constructs to guarantee mutual exclusion and to solve the consumer-producer problem.

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

Table 1: Libc Functions Preloaded (Tiny C Webserver)

4.3 Kmap Network Calls

The most challenging aspect 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. To make Kmap generalizable, one would truly have to intercept every network call (or possible network related).

This task is enormous. Take for example, how services often take advantage of configurable aspects of the network stack like *setsockopt*. Setting options on a particular socket changes the behavior of accept, i/o, and more. For example, our investigation of Flask [4] revealed that it does not call *accept* for new connections. Thus, for Kmap to properly mirror the POSIX API to services in an agnostic way, it would have to model the entire state diagram for sockets.

For our prototype we have focused on HTTP. In our prototype, a service user would contact Envoy with an HTTP GET request. Envoy uses `readv` and `writev` in order to read from and write to multiple file descriptors at the same time. Our tiny C server [9] uses low level file IO system calls (e.g. `read` and `write`). We summarized the intercepted network syscalls in Table-1.

4.4 Initialization

We use the GCC feature *__attribute__((costructor))* to tell GCC to compile our library such that before any functions are invoked we can load kmap. The memory allocation can take a second or two depending on the size of memory allocated. Thus, we run initialization/mapping functions for each side via the constructor, rather than on invocation. This also helps optimize performance by not requiring runtime initialization checks on the functions.

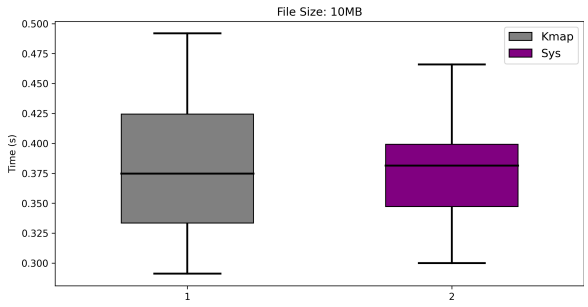

Figure 5: Kmap with Network Under Load

5 Evaluation

Experimental Design: We tested Kmap by running a single docker container with Envoy and our Tiny C Webserver [9]. We then issue a series of programmatic requests to the container, targeting the Envoy port which in turn returns the content from the webserver. We generally served fixed size randomly generated byte files, which we then compared on delivery to confirm efficacy.

5.1 General Performance

Our first test compares the time to transfer of 3 different size files 700B, 100KB, and 10MB. In this experiment we use a buffer of 2^{24} bytes, which can support all the files. Our results are shown in Figure-6. We see that with smaller files Kmap is on par with the default system performance. At higher file sizes Kmap falls behind. Our hypothesis here is that Envoy uses readv to read data out of the buffers into iovectors. Thus, larger files require a large number of reads, and, under the assumption our read compared to the system read is slower, this compounds Kmap's time increase.

5.2 Buffer Size

Our next experiment evaluated the performance of different buffer sizes. We looked at the time to transfer for the tiny file across 2 buffer sizes 2^{16} and 2^{24} and compare those times to the system. The results, shown in Figure-**??**, showed that the 2^{16} Byte buffer performed better. We believe this is because the smaller buffer requires less pages and the memory may be more closely located since the process is not requesting as large a chunk.

5.3 Busy Server

Our previous two experiments were run in isolation without any other running processes. For the majority of microservice deployments this is not representative of the typical workload. Thus, our next experiment simulates concurrent usage. We ran a netcat TCP stream at 5MB/s on the same node as our Kmap test. The results are shown in Figure-5. From this test we see that Kmap's median response is faster than the default system. We view that this reflects that the cost of kernel operations is increased when concurrent other processes are using the network stack as well. Further, this test more closely resembles real-world deployment and thus we are optimistic that Kmap's principles remain strong.

6 Related Work

NetKernel: [8] This work focuses on moving optimizing the network stack by moving it out of the kernel. The impetus was in the VM-age, riping the kernel out of the VM itself provides more control and optimization for users. Their improvements include shared memory. In their work (Section 4.1) they discuss the use of LD_PRELOAD for transport socket redirection. For their usecase, LD_PRELOAD was not

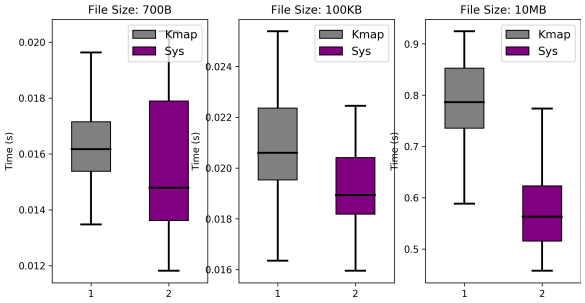

Figure 6: Time to Transfer

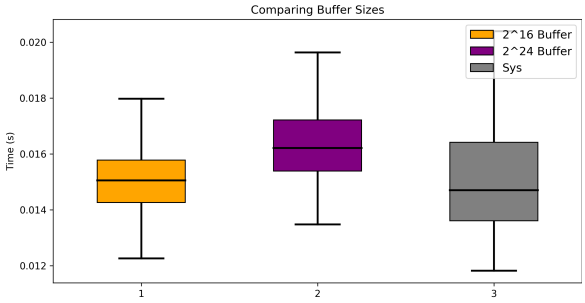

Figure 7: Comparing Kmap Buffer Sizes

robust enough and required too many mappings. However, we see that for Kmap the transport socket redirection should be sufficient.

Slim (OS): [11] SlimOS addresses performance challenges within the container network stack. Similar to Kmap, they attempt to avoid traveling the virtualized network stack and directly preceding through the host network interface. They work works at a lower abstraction level than Kmap but employs a similar principal. SlimOS dynamically links its user-space library onto of containers using LD_PRELOAD on unmodified application binaries.

7 Discussion

Our work in Section-5 shows Kmap is comparable to the default system in certain circumstances and performs better in a select few. Throughout our process, we attempted a number of optimizations which generally did not have much effect. We will highlight them here and explain why we view that they were inconsequential. We will also discuss future directions and learnings from the project.

7.1 Optimizing Kmap

Our results show the end of a number of improvements we made to Kmap. Though no individual improvement significantly adjusted Kmap's performance.

Compiler: Our work began by writing the library with no compiler optimization. Compiling with optimization flags of 02 or 03 did not adjust performance much. The code base is relatively small and low-level so we did not expect much improvement. Nevertheless running compile optimized code is a good practice.

Copying: We also experimented with the way we copied bytes. Our naive first implementation copied at the byte level in and out of the buffer. We improved this by using *memcpy* and even experimented with multithreading *memcpy*. In either case, the improvement we saw was minimal, leading us to our cumulative conclusion that the network stack is highly optimized.

Unsafe execution: Another method of optimization we took was to avoid error checking in the main path of function calls. By this, we mean that we don't null check or offset check as much as is possible. We moved much of our initialization to the library constructor and assume it is run first (start up takes < 1 second). The improvement this technique provided was very slight.

Overall, we feel that these optimizations were relatively surface-level. By this, we mean that they all improve the way we communicate with our shared memory and the other processes. They do not *themselves* optimize shared memory, where we feel is causing the slow-down.

7.2 Next Steps and Lessons

We feel that the work here shows Kmap's promise. Further evaluation and exploration are needed to understand under which environments Kmap performs better. Further paths (i.e TCP/UDP/gRPC) also may show different results. Ultimately, though, we view that this confirms how optimized the network stack is. In principle, Kmap should be a streamlined version of the network stack, however, the decades of work built in to the kernel cannot be unappreciated.

References

[1] Best practices: Benchmarking service mesh performance. https://istio.io/blog/2019/performance-best-practices/.

[2] Correct usage of ld_preload for hooking libc functions. https://tbrindus.ca/correct-ld-preload-hooking-libc/.

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

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

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

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

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

[8] NIU, Z., XU, H., HAN, D., CHENG, P., XIONG, Y., CHEN, G., AND WINSTEIN, K. Network stack as a service in the cloud. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2017), HotNets-XVI, Association for Computing Machinery, p. 65–71.

[9] O'HALLARON, D. tiny web server. http://csapp.cs.cmu.edu/2e/ics2/code/netp/tiny/tiny.c.

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

[11] ZHUO, D., ZHANG, K., ZHU, Y., LIU, H. H., ROCKETT, M., KRISHNAMURTHY, A., AND ANDERSON, T. Slim: OS kernel support for a low-overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 331–344.