Application-tailored Communication with xMesh

Divyanshu Saxena* *UT Austin*
Saksham Goel* *UT Austin*
William Zhang* *UT Austin*
Madhav Tummala* *UT Austin*
Aditya Akella *UT Austin*

Abstract

Current service mesh frameworks have a semantic gap with desired policies. Developers must manually translate high-level policies to service-mesh configurations, which can be inefficient. To address this, we propose xMesh, a framework with M4 language at its core, that provides developers with enough expressiveness, enables conflict detection, optimizes sidecar placement, and enables agile proxy development.

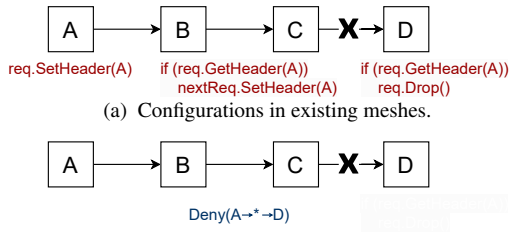
1 Introduction

The transition of applications on the cloud from a monolithic architecture to microservice deployments - where the application is developed in the form of hundreds of *loosely-coupled* services - allows for enhanced scalability, accelerated development cycles, and enables language and framework heterogeneity for various services of the application [8].

However, the microservice architecture introduces additional points of failure and obscures the linear flow of the application logic, sometimes resulting in difficult-to-patch bugs. This has motivated the development of technologies to assist service owners in debugging such as telemetry, tracing, and traffic management. Due to the introduced complexity, these networking policies are difficult to configure, deploy, and update. In addition, these features often require fine-grained control over layer 7 requests. For example, consider a policy that restricts access to an 'experimental' service to users labeled 'beta'. This would require the ingress web server to parse and label 'beta' users' requests with a custom header and would also require each service to check the header and respond accordingly.

Service meshes provide a way to lessen the burden on application developers by abstracting out all microservice communication policies into a separate process that can be deployed *alongside* the application container. This proxy is typically deployed as a container itself, commonly known as the *sidecar*. Several open-source implementations of production-ready service meshes available [4, 5] that use this architecture.

Unfortunately, the current service mesh frameworks are built on imprecise abstractions and hence do not fully relieve developers of their burden. We make three observations as the basis of this claim. (1) The mesh control planes of today expose crude knobs that only allow the developers to provide configurations for one service. For example, consider an access control policy that a request originating at a 'background' service `A` should be denied access to a 'critical' database service `D`, even if it passes through some other services `B, C` on the path from `A` to `D`. In current mesh frameworks, the developer will have to set a custom header at `A`, propagate those headers through the entire path, and then check at `D`. (2) Because developers need to provide separate configurations for each service, these per-service policy configurations may conflict with each other. For example, consider a service `E` that makes a call to service `F`, which has two versions v1 and v2. The developer for `E` may decide to send 20% of its outgoing traffic to v2 of `F`. The developer for `F`, working independently, may decide to impose a load balancing policy to equally divide traffic between the two versions. (3) Mesh frameworks are built atop L7 proxies [2, 3, 5] that expose an API for using them. The mesh control planes today expose their own crude knobs that only have a loose mapping with the features of the corresponding sidecars. Any new features in these L7 proxies take weeks to be lifted onto the control plane and eventually be used by the developer. Specifically, we found that nearly 35% of a total of 4,200 PRs on Istio Proxy's GitHub repo [6] were on upgrading Envoy versions.



(a) Configurations in existing meshes.

(b) Configurations for xMesh.

Figure 1: Complexity in specifying complex application policies in existing mesh frameworks vs. xMesh

Apart from the above three challenges, current service mesh frameworks also impose heavy performance overheads (observation (4)). Today, the *de-facto* approach is to "inject" a sidecar into the microservice graph for each app container. These additional userspace containers significantly increase the memory and CPU usage. In our experiments, we found that using sidecars increases CPU usage by 10-160% and memory usage by 12-35%. Second, in current mesh frameworks, proxies are injected into the microservice graph for each app container, irrespective of whether the application's communication policy requires a sidecar or not. This implies that *every* request received by a service is routed through the sidecar. For L7 proxies that need to perform HTTP parsing, this can be a huge bottleneck [9]. In our experiments, we found that for two popular mesh frameworks, the overhead in P99 latencies was 1.5×-2.5×.

We claim that the lack of precise interfaces between the mesh control plane, the data plane sidecars, and the application communication policies is the root cause of the above problems and a major roadblock in enabling future developments to the mesh dataplane. Inspired by the P4 framework [7], we propose xMesh, a framework that allows developers to specify high-level policies for the application in

*Student at UT Austin

an implementation-independent manner. To this end, xMesh proposes a new service mesh specification language, M4, that fully abstracts the low-level data plane and allows directly expressing high-level policies. M4 solves this challenge by identifying generic entities needed for application policies (e.g. HTTP requests, TCP connections, HTTP connections, state) and identifies the important attributes for these entities that applications care about (e.g. path of a request, or endpoints of a TCP connection).

We intend M4 to be dataplane agnostic. A dataplane is expected to express its capabilities in the form of an M4 *spec* program that provides a specification of the entities it supports, and the associated operations for each entity. These entities can be high-level abstractions on which policies operate. For example, an L7 dataplane may encapsulate a group of HTTP requests selected by the path they have taken in the microservices graph in a single entity. This entity can have operations like `DropRequest()` and `AllowRequest()` that deny or allow the aforementioned requests. The developer can then provide the intended policy in the form of an M4 *policy* program that makes use of these operations. For instance, instead of writing separate programs for both `A` and `D` in the previous example, the developer can directly specify a policy to deny HTTP requests with path $\texttt{A} \rightarrow * \rightarrow \texttt{D}$, which selects every request originating at `A` and ending at `D` (Figure 1). Thus, M4 enables developers to directly express high-level policies (challenge 1). Analogous to P4, the dataplane must also provide a compiler backend that compiles the developer policy down to individual configurations for different sidecars. Any updates to the dataplane need only be reflected in the 'spec' program and can then be directly used by the developers – therefore, avoiding the constant upgrade and feature update process of today (challenge 3).

Since every policy is expressed as an M4 *policy* program, xMesh can identify conflicts in one or more policies specified by the developer (challenge 2). For instance, it can detect that the path $\texttt{E} \rightarrow \texttt{F}$ (to specify the policy to send 20% traffic to v1) and $* \rightarrow \texttt{F}$ (to load balance between different versions) overlap, and their corresponding policies may conflict. Further, this global view can also allow it to choose the minimal set of sidecars needed to implement the policy, avoiding the CPU, memory and latency overheads of extraneous sidecars ignored by current meshes (challenge 4).

Finally, we design an accelerated dataplane for service mesh that can circumvent L7 parsing for a restricted set of policies by implementing the policies as eBPF [1] modules, right on the kernel datapath. Specifically, all policies that do not require header mutations are processed by the eBPF fastpath. All other policies are implemented in a lightweight userspace sidecar. The xMesh control plane can then synthesize programs to be run in eBPF or in the userspace sidecar based on the M4 'policy' program. We envision that the use of M4 programming language can enable easy development of future accelerated proxies, while still keeping the developer interface same.

In summary, we make the following contributions:

- We identify gaps in the interface exposed by current service meshes to the developers. We show how these gaps increase developer burden, introduce potential bugs, and lead to suboptimal policy implementations.
- We carefully design M4, a new mesh specification language that allows dataplane proxies to specify the functionalities they support, and allow developers to write policies without knowing about 'how' the policies are implemented
- We design the xMesh control plane that can parse M4 'spec' and 'policy' programs to identify possible conflicts in policies and synthesize configurations for a minimal set of sidecars, reducing overhead.
- We design an accelerated dataplane for service mesh that is tailored for the application's communication policies. Policies that do not require header mutation are processed in the eBPF fastpath - thereby reducing the dependence on userspace sidecars for all policy implementations.

As future work, we intend to use M4 to specify the capabilities of the accelerated data plane and use it with xMesh to demonstrate benefits over existing mesh frameworks.

References

[1] eBPF. https://ebpf.io/.

[2] Envoy Proxy. https://www.envoyproxy.io/.

[3] HAProxy. https://www.haproxy.org/.

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

[5] Linkerd. https://linkerd.io/.

[6] Pull Requests - istio/proxy. https://github.com/istio/proxy/pulls?q=is%3Apr+is%3Aclosed+envoy%40, 2023.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[8] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[9] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan. Dissecting service mesh overheads, 2022.