Leveraging Service Meshes as a New Network Layer

Sachin Ashok
University of Illinois at Urbana-Champaign

P. Brighten Godfrey
University of Illinois at Urbana-Champaign and VMware

Radhika Mittal
University of Illinois at Urbana-Champaign

ABSTRACT

As modern cloud services have scaled out, applications have moved from relatively monolithic designs to highly modularized fleets of microservices that communicate among each other to perform application-level tasks. These microservices effectively form a network at the application layer, and service mesh frameworks have recently emerged to factor out microservices' common communication functionality.

This paper seeks to highlight the emergence of service meshes as what is effectively a new layer in the networking stack, and the associated new challenges and opportunities. As a case study, we leverage the fact that service meshes can be better informed about application needs, and design a system that utilizes provenance tracing within the service mesh to perform cross-layer prioritization of latency-sensitive requests, within an application serving a mix of workloads. Broadly speaking, we believe that as applications factor out communication into service meshes, an exciting new domain is opening that can utilize techniques from the networking community to improve application performance.

ACM Reference Format:
Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. 2021. Leveraging Service Meshes as a New Network Layer. In *The Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21), November 10–12, 2021, Virtual Event, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3484266.3487379

1 INTRODUCTION

Cloud-based services today are built in functional modules called microservices that carry out narrow roles making them easier to develop in focused teams, to deploy and upgrade one at a time, and to scale out and load balance. For example, a Cisco network monitoring product has hundreds of individual microservices [29] and Uber has over 1000 [44].

To serve any particular request, many of these numerous microservices may be involved across many physical hosts, and they must interact with each other. Thus, microservice architectures imply that network communication becomes more intrinsic to the application's internal functioning. Indeed, inter-microservice communication can be said to form an application-level network, including multiple networking functions such as name resolution, load balancing, and network security. This, however, places a burden on application developers, which has led to the emergence of service meshes. Service meshes lessen the burden on application developers by factoring out microservice communication functions into a separate process called a sidecar proxy. Each application microservice is paired with a sidecar, which performs all inbound and outbound network communication.

The purpose of this paper is to spotlight service meshes as an *emerging new network layer*, and to demonstrate how this layer has opportunities and challenges that differ from lower layers of the stack, but that can be informed by long lines of networking research. In particular, compared to lower layers, service meshes provide more direct visibility, an avenue to understand applications' needs, a point of coordination with lower layers, and easier extensibility. They also lead to new challenges such as performance overhead and cross-stack coordination of shared resources.

We design a case study which demonstrates several of the above opportunities. Sharing infrastructure across tasks with different needs – low latency for real-time and user-facing services, while also serving jobs that mainly require high throughput – is a persistent challenge. Despite many designs at the network layer, deployment has been out of reach of most applications and enterprise networks, in part because these designs require application-specific information. We sketch a design that leverages service meshes to address this problem, through their ability to closely coordinate with applications, and perform cross-layer optimizations within each sidecar proxy. Our early prototype significantly reduces the latency of user-facing requests when the application is processing a mix of workloads.

Overall, we hope this paper will serve as a call to the network community to leverage the emerging opportunities provided by service meshes, and recognize and embrace this new layer in the stack.

HotNets '21, November 10–12, 2021, Virtual Event, United Kingdom

ACM ISBN 978-1-4503-9087-3/21/11…$15.00
https://doi.org/10.1145/3484266.3487379

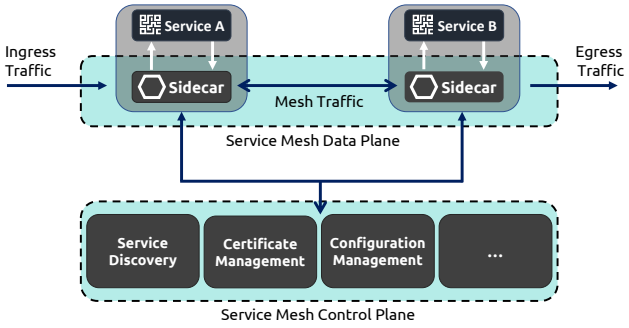

Figure 1: Service mesh architecture, using Istio as an example. Image redrawn from [25].

2 BACKGROUND

Microservice architectures took shape alongside container orchestration frameworks such as Kubernetes [33]. While Kubernetes focused on orchestrating units of compute (containers), it generally did not provide network functionality. App developers would have to implement their own versions of a variety of communication functions, such as security, monitoring, and steering traffic to appropriate microservice instances. This problem became more pressing as microservices architectures matured and were more broadly adopted.

The raison d'être of the service mesh was to factor out microservice communication needs, so a single implementation could be leveraged by many applications. Two early open source service meshes, Linkerd [37] and Istio [24], launched in 2016 and 2017, respectively [36]. Today, Istio is the most widely used, and there are a variety of commercial alternatives [11, 17, 18, 49]. These systems have a generally common set of core functionality, which we sketch next.

In a generic service mesh architecture (Fig. 1), the control plane offers the administrator a centralized location for defining configuration which is then pushed to the individual data plane elements. The control plane also performs certificate generation, service discovery, metric collection, and other functions which in the case of Istio are themselves built as individual microservices.

The data plane is comprised of a set of "sidecar" proxies (called Envoy in Istio). A sidecar is a userspace process running in a container. Each application microservice instance is paired with a sidecar instance, so that all of the microservice's communication is handled via its sidecar – both inbound and outbound, and both internal and external (outside the service mesh) requests. Of course, the same functions could have been provided in a library within the app process, rather than a sidecar. The advantages of a sidecar are similar to the advantages of microservices in general: they facilitate easier upgrades (i.e., the sidecar can be deployed and upgraded without modifying the application) and better modularity of developer teams, and are agnostic to the app's code language and frameworks.

When receiving a request to or from the microservice, the sidecar can perform a variety of functions, such as: service discovery; security in the form of enforcing centrally-defined policies that specify which services are permitted to communicate; traffic encryption; routing requests to appropriate microservice instances; load balancing between replicas; resilience, such as retrying requests and implementing a "circuit breaker" pattern to avoid underperforming instances; and monitoring requests and their key performance metrics.

3 OPPORTUNITIES AND CHALLENGES

Architecturally, we view the service mesh as a new layer in the network stack between application and transport (§3.1). Importantly, this layer comes with new capabilities that have previously been persistent challenges for network infrastructure generally, and for research proposals in particular. We outline some of these opportunities, with research directions that they enable (§3.2-3.5) as well as new challenges (§3.6).

3.1 An architectural perspective

We argue it's useful to think of the service mesh as a new layer in the network stack beneath the application (Fig. 2). In general, a layer is an abstraction that provides functionality used by higher-level modules, and implements that functionality on top of simpler functionality provided by other modules [42]. In the case of service meshes, this layer sits between the application layer and the transport layer.[1] An app built using the service mesh communicates via the service mesh API (so for example, it no longer has to directly open transport connections to remote services). The service mesh API provides relatively high level abstractions like "get the response to this HTTP request from service X". The implementation of this API in the sidecar involves sorting out lower-level details, like exactly where the service X is located, which instance of service X is best to talk to, and how to provide resilience when a connection drops. The sidecar implements these using the lower layer, specifically transport connections to various other processes.

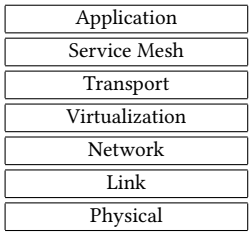

Figure 2: A modern "cloud native" network stack.

[1]In fact, the picture is a bit more complicated. The app and sidecar are usually in different containers, with a transport connection between them. However, the app-to-sidecar connection can be seen as an architecturally-negligible detail which could have been a library call, as indeed it is in Netflix's software suite [6]. Either way, the sidecar still ends up taking an HTTP message from the app layer as input.

This view does not fit the specific factorization of layers defined in the OSI 7-layer model. OSI layers 5 and 6 (session and presentation) also sit between the transport and application layers. Service meshes share some features with the session layer (e.g., connection establishment and restart) and the presentation layer (e.g., encryption). However, neither layer is a good description of a service mesh, which provides other features like security policy enforcement and performance-aware load balancing. Our view is that many different layered architectures are possible, and the layering used in practice has evolved over time. The Internet itself never exactly matched the OSI model, and has evolved in other ways – for example, cloud network virtualization arguably forms a layer of its own [43, 46, 48] – and service meshes are another evolving layer.

A service mesh could alternately be viewed as simply another set of microservices running in userspace alongside the application. This description is correct but incomplete: it misses that the service mesh abstracts out communication functionality. That abstraction, which is a property of layering, helps highlight the opportunity to utilize service meshes to observe and enhance communication across the whole application. We discuss some of these opportunities next.

3.2 Better visibility

Function calls in a monolithic architecture in many cases become network communication in the microservice architecture. This can reveal a rich picture of the application's internal operations, even with purely passive observations. As the layer directly below the application, the service mesh is ideally positioned to capture this information. Lower layers, like the physical network, can also observe more about a microservice-based application than a monolithic one. However, lower layers will have less visibility than the service mesh due to lacking context of API calls, encryption, and because the service mesh can coalesce multiple requests into a single transport-layer flow (among other reasons).

This visibility could be used for monitoring, troubleshooting, and root cause analysis. For example, [4] described a performance telemetry system for Gmail and certain other Google services which tracks the distributed call tree, and performs "coordinated bursty tracing" across multiple layers of the stack in a coordinated interval of time. Such techniques might be adaptable to service meshes, via distributed tracing already implemented in service meshes [23] and leveraging sidecars to trigger cross-layer logging, thus making [4] or similar proposals available to a wider swath of applications.

3.3 Better knowledge of application needs

Many enhancements to network performance, such as priority-aware flow scheduling [8, 19, 51], require some knowledge of application needs. As they are positioned close to the application in user space, service meshes may be able to obtain this knowledge. For example, service meshes directly handle HTTP-level service requests and responses, and are informed of a *set* of acceptable destinations (replicated microservices in a load balancing pool) rather than just a single IP address. Applications could also directly signal preferences via the app-to-sidecar API.

This suggests a specific research direction: utilizing knowledge of app needs or objectives in the service mesh to inform and control lower layers of the stack. For example, knowledge of flow priority or sizes could help optimize transport, flow scheduling, and network-layer traffic engineering (TE).

The service mesh API can allow for explicit signaling of this information from the app, but some apps may not provide it. Thus, another open problem is to automatically infer what's best for the application, by leveraging the app information innately available to the service mesh (e.g., request sizes, headers, response time statistics) and perhaps the ability to run "in vivo" experiments on a small fraction of requests. (Inference of app needs can also be seen as an important special case of better visibility; §3.2.)

3.4 Easier evolvability

Adding and modifying functionality in a deployable way is a recurring difficulty for network infrastructure protocols and systems. One reason is that many other systems are built on top of the network infrastructure and "bake in" assumptions about its behavior, and this could be true of service meshes as well. However, service meshes have several distinct advantages. First, they run in user space on hosts, making them more accessible compared to changes to the kernel, network virtualization, or physical network.[2] Second, service meshes communicate with the layer above (i.e., apps) via APIs which are easily extensible with optional parameters. At lower protocol layers, even a simple change like adding a header field often involves careful work and perhaps standardization processes, and coordinating these changes with higher layers can be even harder in practice.

This extensibility suggests an exciting direction of leveraging service meshes as a place to implement network functionality that has been otherwise hard to deploy within the network. Numerous recent congestion control and transport protocol proposals [5, 13, 28, 35, 39] could be utilized in the sidecar-to-sidecar channel while leaving the application itself unmodified. Other examples implementable in the sidecar could include adaptive replica selection [30] and issuing redundant requests [50] to cut tail latency.

[2]Indeed, even with advances in programmable switching, changes to most network infrastructure are completely inaccessible to most software developers (consider, for example, users of public cloud).

3.5 Coordination with lower layers

Just as the service mesh can benefit from knowledge of the application, it may also benefit from knowledge of lower layers. For example, a physical network SDN controller could provide information about the level of congestion along network paths, and the service mesh could use this to control request rates or adjust load balancing among service instances. A more advanced optimization would be to jointly optimize network TE and sidecar load balancing.

At the transport layer, [15] proposed a related method: using an SDN controller's information about link utilization to adjust TCP parameters. Such cross-layer coordination, especially between a host and a network operator, is usually difficult to deploy but may be made possible through service meshes due to their easier evolvability (§3.4), and potentially the ability to leverage the service mesh's central controller as a more convenient point of interface. In addition, the service mesh has richer control mechanisms (e.g., selecting among replicas) that could lead to more optimal solutions.

3.6 New challenges

In addition to opportunities, service meshes introduce new problems. Among the most obvious is the increased latency imposed by the two sidecars interposed between each application-layer end-to-end communication, which is in the range of 3 msec at the 99th percentile for Istio [26]. While this is acceptable for many applications, it could be costly for latency-sensitive apps involving tens of hops among microservices. Recent research on low-latency stacks [7, 16, 21] could have an impact here.

As a network in itself, the service mesh covers many functions that are similar to traditional network tasks, such as routing and load balancing, transport, and backpressure, leading to several challenges. The right algorithms for these modules may be non-obvious. Here, experience in the networking community may be relevant; for example, Structured Streams Transport [13] could assist the sidecar in multiplexing many requests over a single transport connection. Second, these functions which are similar to lower layers, may in fact be controlling overlapping sets of resources, leading to potential conflicts. Verification has been proposed for this subproblem [38] but a principled understanding of how to safely decompose dynamic control is needed.

4 CASE STUDY: CROSS-LAYER PRIORITIZATION

To make the ideas of §3 more concrete, we present a case study: providing request-level prioritization in a microservice-based application, via cross-layer optimizations. Our design takes advantage of several of the opportunities outlined above: better knowledge of application needs, easier evolvability, and cross-layer coordination.

4.1 Motivating Scenario

Consider a hypothetical microservice-based e-commerce application. The app serves user requests where fast response is critical. We refer to this as the *latency-sensitive* workload: response times on the order of roughly 200 ms are desirable. At the same time, analytics workloads scan large sets of data to optimize advertising and user product recommendations, and other jobs write periodic updates to the product database and collect logs for system monitoring. Unlike the latency-sensitive workload, it is acceptable to delay these tasks by minutes or perhaps even hours. However, all tasks share many of the same microservices (e.g., caches and databases), sometimes buried several hops deep in the tree of API calls.

In such scenarios, the main method of achieving high utilization while also satisfying latency-sensitive services is to prioritize the latter. This can apply to compute, network, or storage bottlenecks. Focusing here on the network (which can be a significant part of the problem, especially with multi-tiered microservice-based apps [14]), a large body of work has explored prioritization, including via traffic priority classification, flow scheduling, "coflow" scheduling, routing, or combinations of these [2, 8, 9, 19, 20, 27, 51]. However, deployments have been limited. Large providers, notably Google and Azure, have implemented systems that use explicit knowledge of internal applications and extensive customization of the physical network to prioritize classes of applications in inter-datacenter WANs [20, 27]. Outside of these, app-informed prioritization is not generally available to users of enterprise data centers or public cloud. Even though the low-level mechanism of prioritized queues is supported in OSes and off-the-shelf physical switches, mechanisms are lacking to automatically understand application needs and to appropriately dynamically control lower layers of the stack, in a deployable way.

4.2 Design

We sketch the design of a system that leverages and extends service meshes to optimize request processing for latency-sensitive and latency-insensitive tasks. The basic idea includes three components:

(1) Classify applications' performance objectives at the ingress point of the request.
(2) Carry these performance objectives through the entire system with each request, via application-level tracing through the service mesh.
(3) Implement cross-layer optimizations, such as:

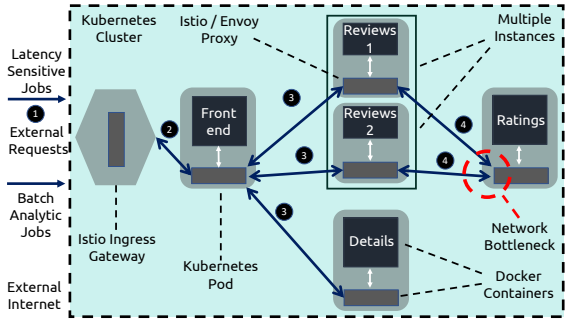

Figure 3: The e-library microservice running Istio.

(a) Service mesh: request-level prioritization and improved traffic routing among service instances.
(b) Transport: utilization of *scavenger* transport protocols [34, 39, 45] for latency-insensitive requests.
(c) OS/hypervisor: packet-level prioritization of latency-sensitive flows at the (virtual) NIC.
(d) Physical network: Flow scheduling and TE with awareness of packet priority.

Optimization (d) would involve the service mesh supplying knowledge of flow priority to the physical network, either in-band (packet tagging) or out-of-band (an API call into the SDN controller). The physical network can then schedule and route flows better, similar to [20, 27].

This design illustrates some of the advantages described in §3. First, it uses knowledge of application needs through tracing the provenance of requests. Second, it uses the easily evolvable sidecar proxy to make changes, including across layers, for example by modifying the sidecar-to-sidecar transport protocol and OS packet handling without modifications to the application. Finally, the overall architecture takes advantage of the service mesh being its own layer of the stack, so the benefits of this functionality can apply to many apps without app developers needing to understand the details.

4.3 Prototype

Setup. We run an e-library microservice (based on Istio's sample bookinfo application [22]) on a single 32-core Intel Xeon Silver server running Ubuntu 16.04. We orchestrate the microservice using Kubernetes KIND [32]. All inter-pod communication traverses through 15 Gbps emulated links, except for a single bottleneck set to 1 Gbps. Kubernetes co-locates all containers within a pod on the same host, and hence intra-pod communication goes through the localhost.

As shown in Fig. 3, the ingress gateway of the microservice routes the incoming (external) requests to the application front end (stage 1 and 2). The front-end processes the requests and, to serve them, spawns internal requests to other components (stage 3). Requests propagate through the application as per the request tree (stage 4), and the responses propagate back to the end-user.

We use the wrk2 [47] load generator to generate two different workloads that hit the ingress gateway simultaneously: (i) latency sensitive requests representing users traversing a website, and (ii) latency-insensitive requests (≈ 200× larger) representing a batch analytics job. We use uniformly random inter-arrival times for both, with average request per second (RPS) levels ranging from 10 to 50 across experiment runs. Each experiment run lasts 5 minutes excluding warm-up and cool-down periods. The aforementioned bottleneck is set between the reviews and the ratings components, so network responses of both workloads compete for bandwidth here.

Implementation. Our prototype implements a version of the three design components (§4.2) as follows.

(1) The application handling requests at the ingress (front-end) sets a custom HTTP header field [40] indicating either low or high priority. After processing this request, it attaches the same priority bits onto the further internal requests it spawns and sends to its sidecar.
(2) Each sidecar, in turn, propagates the priority of an incoming request by copying its priority header onto the associated responses and outgoing requests. The sidecar knows which outgoing requests map to which incoming ones because they have the same global request ID (x-request-id [23]) which is propagated to those requests by the application to enable existing service mesh functionality.[3]
(3) To provide distinct service to different kinds of requests, sidecars forward them to either a high or low priority pod (in our case, front end forwards requests to either reviews replica 1 or 2 depending on priority). We then prioritize these flows' packets. We use a very simple method in this prototype. Specifically, we set Linux TC rules that direct packets matching the pod's IP address to be given nearly-strict prioritization (up to 95% of bandwidth) in the kernel's outgoing packet queue on the sidecar container's virtual interface. We leave other optimizations to future work.

Results. Cross-layer prioritization significantly (≈ 1.5×) improves the p50 and p99 response latency of the higher priority latency-sensitive workloads (Fig. 4). This improvement comes at the cost of degrading the performance of the latency-insensitive workloads (less than 5% increase in the p99 response latency) The absolute numbers, of course, will depend on the workload distributions; this basic test illustrates that significant improvements are possible.

[3]Service meshes use various custom HTTP header fields to communicate with the app. For example, the distributed tracing feature uses such metadata to tie together numerous trace *spans* (i.e., metadata about a request's execution within one microservice instance) to create a distributed trace [23].

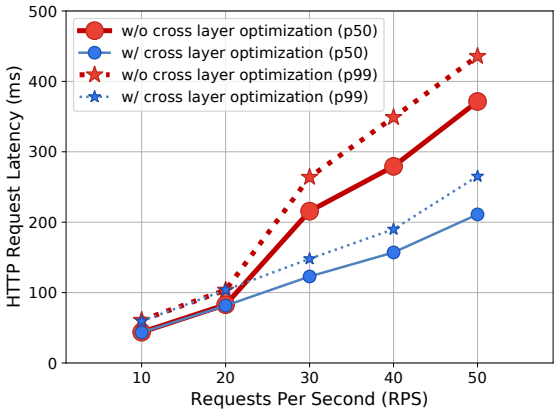

Figure 4: Reduction in request latency from cross-layer optimization.

5 DISCUSSION

Can't all this just be implemented in the application? In principle, yes, and this is true not only of our enhancements, but also of the features of the service mesh itself. Service meshes exist because app-based implementations place a needless burden on developers, at best. Worse, per-app implementations could result in inconsistent security policies, more frequent bugs, and missed opportunities. These considerations are especially true of the directions we propose, which involve specialized networking knowledge (e.g., performance tradeoffs of TCP variants) outside the scope of most app development teams. Cross-layer coordination and other advanced functions don't become *easy* in a service mesh, but by modularizing the functionality and amortizing the cost across many applications, they may become *viable*.

Lack of a single service mesh. The service mesh space is somewhat fragmented, given the numerous open source and commercial products. While this space is evolving, we note that universal deployment is not needed for most of the directions we propose. Also, standardized interfaces like the Service Mesh Interface (SMI) project [12] are emerging.

Maturing cross-layer prioritization. Our prototype (§4) can be extended, e.g., by coordinating management of other resources beyond the network (i.e., compute and storage), allowing the application to specify more fine-grained preferences, and leveraging other optimizations such as prioritized request queuing and improved transport protocols.

Co-designing with Network Service Meshes. A Network Service Mesh (NSM) [10] is a complementary solution offering control and configurability of L2/L3 functionality at hosts, such as firewalls, VPN tunnels, and forwarding frames (as opposed to the higher level functionality that service meshes provide). Today, service meshes and NSMs both run independently but a co-design could offer broader control.

6 RELATED WORK

Service meshes bear similarity to virtualized networks [31] which provide isolated environments for cloud tenants. Both arguably form a network layer (Fig. 2), utilize SDN-style centralized control planes, and utilize host-based data planes. Service meshes differ in that they communicate with apps more closely via APIs, operate above the transport layer, and are under the control of software developers (whereas network virtualization is under the control of the cloud provider). These properties directly influence the opportunities in §3.

Distributed stream processing systems (e.g. [1]) involve directed acyclic graph of processing elements that continuously process data. These elements are essentially microservices, and streaming systems provide communication between them with some functionality that overlaps with service meshes, like fault tolerance and load balance. In general, stream processing and more recent distributed data processing frameworks [41] may also be attractive locations to implement some of the directions we proposed, but service meshes have advantages in providing a more general communication abstraction separated from the application.

One of the few papers viewing service meshes from the perspective of network architecture is [3]. Observing that service meshes can be viewed as an SDN at the application layer, [3] sketched a "full stack SDN" wherein a single virtual switch could process traffic at a range of protocol layers L2-L7. Such a design could offer interesting opportunities for implementing cross-layer optimizations, but is overall complementary to the research directions we have discussed.

7 CONCLUSION

This paper makes the case that as modern cloud-native software is becoming more like a network, the service mesh is emerging as a distinct layer in the network stack, with new challenges and opportunities. Our prototype demonstrates a concrete example of leveraging these opportunities to improve the response time of user requests in a system with a mix of workloads. In closing, we note that even though the "new action" that we highlight begins with changes in software near the top of the stack, the implications of those changes cut across layers, from the physical network to the application – as do many of the specific open research directions we outline. We therefore believe that networking research has an impactful role to play that is all the more exciting for its direct relevance to application needs.

Acknowledgements. We thank our shepherd, Anees Shaikh, and the anonymous reviewers for their valuable feedback. This material is based upon work supported by Intel, Facebook, and AG NIFA under Grant No. 2021-67021-34418.

REFERENCES

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The design of the Borealis stream processing engine. In *Cidr*, Vol. 5. 277–289.

[2] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2018. Sincronia: Near-Optimal Network Design for Coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3230543.3230569

[3] Gianni Antichi and Gábor Rétvári. 2020. Full-Stack SDN: The Next Big Challenge?. In *Proceedings of the Symposium on SDN Research (SOSR '20)*. Association for Computing Machinery, New York, NY, USA, 48–54. https://doi.org/10.1145/3373360.3380834

[4] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance analysis of cloud applications. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 405–417.

[5] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical delay-based congestion control for the internet. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 329–342.

[6] Samir Behara. 2019. Microservices Journey from Netflix OSS to Istio Service Mesh. (2019). https://dzone.com/articles/microservices-journey-from-netflix-oss-to-istio-se

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*.

[8] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. 31–36.

[9] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 443–454.

[10] CNCF. 2021. Network Service Mesh. (2021). https://networkservicemesh.io/

[11] CNCF. 2021. Open Service Mesh. (2021). https://openservicemesh.io

[12] CNCF. 2021. Service Mesh Interface. (2021). https://smi-spec.io

[13] Bryan Ford. 2007. Structured streams: a new transport abstraction. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*. 361–372.

[14] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. 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)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[15] Monia Ghobadi, Soheil Hassas Yeganeh, and Yashar Ganjali. 2012. Rethinking end-to-end congestion control in software-defined networks. In *Proceedings of the 11th ACM Workshop on Hot Topics in networks*. 61–66.

[16] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*.

[17] HashiCorp. 2021. HashiCorp Consul. (2021). https://www.consul.io

[18] Red Hat. 2021. Red Hat OpenShift Service Mesh. (2021). https://www.openshift.com/learn/topics/service-mesh

[19] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. 2012. Finishing Flows Quickly with Preemptive Scheduling. In *ACM SIGCOMM*.

[20] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. 15–26.

[21] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. In *OSDI*.

[22] Istio. 2021. bookinfo. (2021). https://github.com/istio/istio/tree/master/samples/bookinfo

[23] Istio. 2021. Distributed Tracing Overview. (2021). https://istio.io/latest/docs/tasks/observability/distributed-tracing/overview/

[24] Istio. 2021. Istio. (2021). https://istio.io

[25] Istio. 2021. Istio Architecture. (2021). https://istio.io/latest/docs/ops/deployment/architecture/

[26] Istio. 2021. Performance and scalability. https://istio.io/latest/docs/ops/deployment/performance-and-scalability/. (June 2021).

[27] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 3–14.

[28] Nathan Jay, Noga H. Rotman, P. Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2018. Internet Congestion Control via Deep Reinforcement Learning. In *NeurIPS Deep Reinforcement Learning Workshop*.

[29] Vimalkumar Jeyakumar, Omid Madani, Ali Parandeh, Ashutosh Kulshreshtha, Weifei Zeng, and Navindra Yadav. 2019. ExplainIt!–A declarative root-cause analysis engine for time series data. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. 333–348.

[30] Gauri Joshi, Emina Soljanin, and Gregory Wornell. 2017. Efficient redundancy techniques for latency reduction in cloud systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 2, 2 (2017), 1–30.

[31] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. 2014. Network virtualization in multi-tenant datacenters. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 203–216.

[32] Kubernetes. 2021. kind. (2021). https://kind.sigs.k8s.io/

[33] Kubernetes. 2021. Kubernetes. (2021). https://kubernetes.io/

[34] Aleksandar Kuzmanovic and Edward W Knightly. 2006. TCP-LP: low-priority service via end-point congestion control. *IEEE/ACM Transactions on Networking* 14, 4 (2006), 739–752.

[35] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*. 183–196.

[36] Frederic Lardinois. 2017. Google, IBM and Lyft launch Istio, an open-source platform for managing and securing microservices. (May 2017). https://techcrunch.com/2017/05/24/google-ibm-and-lyft-launch-istio-an-open-source-platform-for-managing-and-securing-microservices/

[37] Linkerd. 2021. Linkerd. (2021). https://github.com/linkerd/linkerd2

[38] Bingzhe Liu, Ali Kheradmand, Matthew Caesar, and P. Brighten Godfrey. 2020. Towards Verified Self-Driving Infrastructure. In *Nineteenth ACM Workshop on Hot Topics in Networks (HotNets)*.

[39] Tong Meng, Neta Rozen Schiff, P. Brighten Godfrey, and Michael Schapira. 2020. Proteus: Scavenger Transport And Beyond. In *ACM SIGCOMM*.

[40] Keith Moore. 1996. MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text. RFC 2047. (Nov. 1996). https://doi.org/10.17487/RFC2047

[41] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 561–577.

[42] Larry Peterson and Bruce Davie. 2019. *Computer Networks: A Systems Approach, version 6.2-dev*. Elsevier. https://github.com/SystemsApproach/book

[43] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. 2009. Extending Networking into the Virtualization Layer. In *Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*.

[44] Matt Raney. 2016. What I Wish I Had Known Before Scaling Uber to 1000 Services. In *GOTO Chicago*. https://www.youtube.com/watch?v=kb-m2fasdDY

[45] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. 2012. Low Extra Delay Background Transport (LEDBAT). In *RFC 6817*.

[46] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. 2009. *FlowVisor: A Network Virtualization Layer*. Technical Report. Deutsche Telekom Inc. R&D Lab, Stanford, Nicira Networks.

[47] Gil Tene. 2019. Wrk2: a HTTP benchmarking tool based mostly on wrk. (2019). https://github.com/giltene/wrk2

[48] VMware. 2021. Network Virtualization. (2021). https://www.vmware.com/topics/glossary/content/network-virtualization

[49] VMware. 2021. VMware Tanzu Service Mesh. (2021). https://www.vmware.com/products/tanzu-service-mesh.html

[50] Ashish Vulimiri, Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. 2013. Low latency via redundancy. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. 283–294.

[51] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/2018436.2018443