Application Defined Networks

Xiangfeng Zhu[1] Weixin Deng[1] Banruo Liu[2] Jingrong Chen[3] Yongji Wu[3]
Thomas Anderson[1] Arvind Krishnamurthy[1] Ratul Mahajan[1] Danyang Zhuo[3]
[1]*University of Washington* [2]*Tsinghua University* [3]*Duke University*

Abstract

With the rise of microservices, the execution environment of many cloud applications has become a set of virtual machines or containers connected by a flexible and feature-rich virtual network. We argue that the implementation of such virtual networks should be completely application-specific and not layered on top of general-purpose network abstractions from the Internet age. Such layering tends to more than double the latency and CPU usage of applications. We propose *application-defined networks* in which developers specify network functionality in a high-level language and a controller generates a custom distributed implementation that runs across available hardware and software resources. Experiments with a preliminary prototype suggest that, compared to the state of the art, ADN reduces latency by up to 20x and increases the throughput by up to 6x.

CCS Concepts

• **Networks** → **Programming interfaces**; **Cross-layer protocols**;

Keywords

Application Networking, Service Mesh, Microservices

ACM Reference Format:
Xiangfeng Zhu, Weixin Deng, Banruo Liu, Jingrong Chen, Yongji Wu, Thomas Anderson, Arvind Krishnamurthy, Ratul Mahajan, Danyang Zhuo. 2023. Application Defined Networks. In *The 22nd ACM Workshop on Hot Topics in Networks (HotNets '23), November 28–29, 2023, Cambridge, MA, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3626111.3628178

1 Introduction

Since the dawn of the Internet, the design and implementation of data networks has valued generality—the ability to support a wide range of applications—and has used a modular organization to meet this goal in a practical manner. The Internet architecture is organized as a layered stack of protocols. Each protocol offers a specific functionality, building atop one or more lower-layer protocols.

Generality and modularity, however, impose bandwidth, compute, and latency overhead [27]. Application messages may be wrapped first in HTTP, then in TCP, and then IP, and are processed in sequence by multiple protocols at the sender and the receiver. Even so, the general network often cannot support all the requirements of a given application. The result is that it does too much for some applications (at a high cost) and too little for others [1, 2, 30, 38, 56]. For instance, many distributed applications need load balancing across replicas, which the Internet does not provide, forcing applications to engineer their own solutions via middleboxes.

High overhead and imperfect application support may be inevitable for a general network, but many networks today are built to support a single application. The key driver for such application networks is microservices [29], where application logic is split across many (sometimes thousands [47, 65]) services. Communication between microservices has rich requirements, such as load balancing, rate limiting, authentication, access control, and telemetry. Engineers use service meshes such as Istio [19] and Linkerd [21] to build networks that meet these requirements. These networks are virtually isolated and have specific ingress and egress points to communicate externally. Application networks are widespread, in use at 90% of organizations that develop cloud applications [13].

The tragedy of today's application networks is that, even though they serve a single application, they are built using the same abstractions designed for general-purpose communication. Service meshes assume that applications emit IP packets that contain other standard protocols (e.g., TCP, HTTP, and gRPC). A local proxy intercepts these packets and, in the manner of middleboxes, parses and unwraps the network packets. It then applies the network policies and wraps the packets again before sending them to the receiver. The receiver has a local proxy as well, which also unwraps the packet, processes it, and wraps it again before handing it off to the application.

This work is licensed under a Creative Commons Attribution International 4.0 License.
HotNets '23, November 28–29, 2023, Cambridge, MA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0415-4/23/11.
https://doi.org/10.1145/3626111.3628178

This architecture of service meshes has significant downsides. Depending on the configuration, it can increase message processing latency by up to 2.7-7.1x and CPU usage by up to 1.6-7x [3, 9, 12, 52, 66]. Layering also hides or obscures information, which makes it hard to implement application-specific network policies (e.g., choosing replicas based on information in the application's RPC) [4, 14]. Finally, being general, service mesh implementations are large and complex, so it is almost impossible to accelerate them via programmable kernel, NICs, and switches [28, 49, 50, 60].

We argue for application networks to be fully customized to the application and its deployment environment. The key challenge in realizing this vision is: how to enable custom application networks without excessive burden on each application's developers to implement their own network functionality? We propose to address this challenge via *Application Defined Networks.* ADNs run atop a network with basic layer-2 connectivity, similar to that provided by cloud virtual networks. Anything else that the application needs is expressed in a high-level, domain-specific language. We orient the specification language around processing RPC messages emitted by the application because that processing is most relevant [37, 59, 64]. A compiler takes this specification and generates an efficient, distributed implementation across available hardware and software resources, and a runtime controller dynamically reconfigures the network based on workload and failures.

Decoupling the specification of the network functionality from its implementation allows us to generate implementations customized for the application and avoid the fundamental trade-off between doing too little or too much that any general-purpose implementation must make [56]. Further, because we know the semantics of network processing, we can apply optimizations such as selectively offloading network functions to hardware and parallelizing or reordering them while preserving semantics. It also allows us to scale network processing without disruption, as the number of microservice instances changes or the workload scales.

We develop a preliminary ADN prototype to evaluate our proposed approach. It instantiates the desired network functionality by auto-generating modules of mRPC [25]. Our experiments show that ADN can reduce the end-to-end RPC latency by 17–20x and increase the RPC throughput by 5-6x compared to implementing the same functionality using Envoy [18], the most popular proxy for service meshes. We also find that, compared to hand-optimized implementations, the overhead of generated implementations is only 3–12%, while reducing the lines of code by two orders of magnitude.

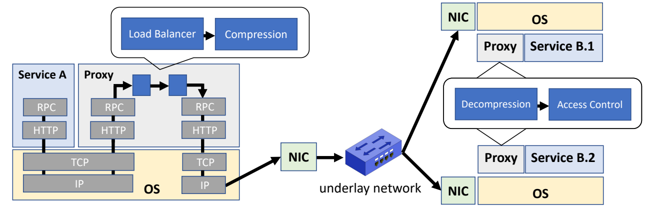

Figure 1: Packet processing in service meshes.

2 The curse of generality

We highlight the pitfalls of building application networks with general abstractions using an example. We view applications as sources and sinks of RPC messages and the "network" as everything that happens to RPCs between application send and receive. Consider an application with two microservices, A and B. Service B is shared, and its two instances, B.1 and B.2, hold a subset of the object identifier space. The application developer wants the network to 1) load balance RPC requests from A to B.1 or B.2 based on the object identifier in the request, 2) compress and decompress the RPC payload, and 3) perform access control based on user and object identifiers in the RPC request.

One could implement these network policies along with the application code itself, but that is not practical. Network policies often evolve independently from the application logic, and it is not practical to modify the application source and re-deploy each time they change. Further, for trust issues, some network policies (i.e., access control) must be enforced outside the application. Thus, the developer needs to implement the network outside of the application, even though it is meant to serve only this application.

Preferring generality, the application developer today does not use a custom request processor that could inspect and manipulate the message to achieve the desired policy. Instead, they lean on a standardized protocol, say HTTP, that allows arbitrary information to be embedded in its headers and modifies the application to add headers for object and user identifiers. Because they choose HTTP, TCP and IP are also chosen as additional layers into which application information is wrapped. Our application does not care about these layers otherwise.

Then, the developer selects a module that can enforce their policies; this functionality is common in L7 proxies [7, 11, 18]. Finally, they need a mechanism such that the application's traffic reaches this module when sent to B. This can be accomplished by intercepting and rewriting the IP packets (e.g., using *iptables*) generated by the application or using DNS to resolve B.1 and B.2 to the address of the module. Once the routing module gets the packet, it parses it to extract the HTTP header and sends it along to the right version of B.

Figure 1 shows the resulting packet path and processing. The application RPC library serializes the request message, and the kernel network stack (configured by iptable rules) forwards the message to the proxy, which typically needs to parse the message headers and deserialize the payload to enforce the desired policy. The proxy then re-encodes the headers and re-serializes the message for transport.

Service meshes [19, 21] today follow this architectural paradigm. The proxies are called sidecars, and they run as a separate user-space process (or container), intercepting and manipulating all incoming and outgoing packets. The key advantage of this approach is that it can support a range of applications, but it has significant downsides too.

High overhead. Packets travel up and down the stack, and they are encoded and decoded multiple times. This has high overhead in terms of application latency and server CPU. SPRIGHT shows that service meshes can reduce throughput, increase latency, and increase CPU utilization by 3-7x [52] (on top of an already high baseline [42, 51]). A dominant component of service mesh overhead is parsing all the protocol headers to recover wrapped information [66]. They also sometimes implement functionality that overlaps with that of lower layers (e.g., retries, rate limiting) because the application desires different semantics [24].

Non-portability. With service meshes, developers implement desired network behaviors by choosing and chaining specific software plugins such as load balancers and loggers. Such network functions can only run within the context of the sidecar and use vanilla IP for transport. This limitation runs up against the increasingly programmable nature of the OS kernel (via eBPF), and the availability of programmable networking hardware (NICs and switches). Parsing and processing for many standardized protocols are almost impossible to offload to kernel [49, 62] or hardware [28, 50, 60]. As just one difficulty, using programmable networking hardware often requires custom header designs due to hardware constraints [41, 45]. A P4-based programmable switch has access to about the first 200 bytes of each network packet [63]. To offload load balancing, we must put the field the load balancer needs into the first 200 bytes of the packet, which may not happen with multiple layers of header wrapping.

Poor extensibility. High overhead and non-portability of current application networking architecture meshes might be more tolerable if they were highly extensible, but that is not the case. Network policies that are hard to express using standard protocols are hard to build and deploy. Consider a request routing policy that sends RPC requests of type T2 to a specific service instance, but only when it follows an RPC of type T1. For such custom functionality, service meshes offer a plugin framework. However, low-level abstractions used for such plugins (IP or HTTP packet, not RPCs) make

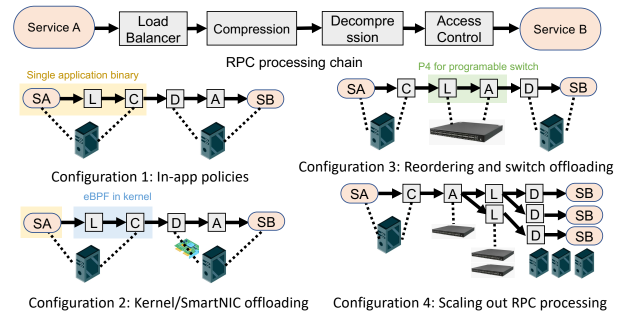

Figure 2: Possible realizations of a RPC processing chain (described in §2).

them hard to develop [4, 14] and the isolation mechanisms for safely running these plugins (e.g., Web Assembly) further drive up the overhead [66].

3 Application defined networks

Given the pitfalls of building application networks using general-purpose abstractions and implementations, we advocate building them in a way that is fully customized to the application and its deployment environment. The network and the software stack under the application should offer no protocols or abstractions by default except for a (virtual) link layer that can deliver packets to endpoints based on a flat identifier such as a MAC address. Cloud virtual networks (e.g., AWS VPCs [17]) provide this abstraction, and technologies like VXLAN [16] can implement it anywhere.

Everything else that the network does is specified by the application developer in a domain-specific language (DSL). We propose to structure this specification as a chain of *elements*, each an operation on an RPC message between two services. A controller decides how to realize the specification in the application's deployment environment. Depending on available resources, RPC processing may happen in the RPC library (e.g., gRPC), in-kernel (e.g., using eBPF), in a separate process (as today), on a programmable hardware device, or in a mix of locations. The controller can also opt to run multiple element in parallel or reorder them.

Figure 2 shows how a controller may realize the desired RPC processing described in §2 in different deployment environments. Configuration 1 shows the case where it deploys the load balancer and compression as part of the RPC library (akin to gRPC proxyless [6]). Configuration 2 moves these functions to the OS kernel on the sender side and to a SmartNIC on the receiver side. Configuration 3 moves load balancing and access control to a programmable switch and also reorders the processing after automatically determining that reordering preserves semantics. In this example, not compressing the RPC field that the following load balancer uses is enough to preserve semantics. Configuration 4 replicates RPC processing to increase throughput.

The exact choice of configuration depends on (1) resources available in the deployment environment, (2) the security model (e.g., mandatory RPC policies should not be enforced inside the same application binary), and (3) the current workload. Our main observation is that once we have a high-level specification of desired network behavior, we can automatically generate a highly efficient implementation. How the RPC message is packaged on the wire and what headers are needed are also automatically determined.

4 Key Research Questions

Realizing the ADN concept requires answering a few key research questions.

Q1: *What abstractions should our DSL provide to specify RPC processing?* The abstractions should be high-level, independent of the underlying platforms, while being amenable to efficient implementation. They should also 1) allow a range of automatic optimizations such as re-ordering, offloading, and generating minimal headers; and 2) enable reasoning about the internal state of elements because that is key for seamless migration and scaling [32].[1]

We also want to enable developers to reuse code of elements developed by others, instead of having to implement their own each time. Element reuse needs careful consideration because there are no standard headers (like HTTP), and an element that manipulates an RPC field of one application may not necessarily work in another. Finally, we should allow developers to specify message ordering and reliability constraints and any element location constraints (e.g., the encryption element must be co-located with the sender).

Q2: *How to translate the high-level specifications to efficient distributed implementation across a range of hardware and software platforms?* This includes both the low-level code (e.g., eBPF, P4) and packet header design for cross-device communication. When multiple elements run on the same device, we should be able to do cross-element optimizations. Finally, we need to determine the minimum set of headers needed to satisfy the network requirements.

Q3: *How to determine the location(s) where network processing happens across available resources and expand/collapse processing based on workload, without disrupting applications?* When a new application is deployed, the ADN controller needs to pick an initial configuration based on the specification and available resources. Once the application has been running, it may need to reconfigure (e.g., picking a configuration in Figure 2) according to the current workload. When the workload increases, we may need to scale out the RPC processing chains onto more compute devices. Such reconfigurations should not disrupt the application.

[1]Specifying network processing using chained elements is not new. A seminal system is Click [39] which enables building modular packet processing pipelines. Packet processing is specified as a directed graph of elements, where each element is C++ code that can use any C++ data structure. Because of these design choices, Click elements are hard to offload and hard to migrate and scale up/down [32, 53].

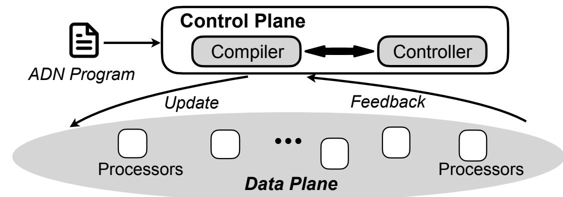

Figure 3: ADN Architecture.

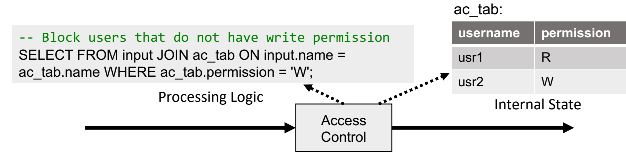

Figure 4: An element that implements access control.

5 A Potential Approach

We outline a potential approach to realizing ADN, which answers the aforementioned questions. Figure 3 shows the architecture. The input program describes the network functionality as a chain of ADN elements in our DSL. The control plane includes a compiler and a runtime controller, while the data plane consists of hardware or software-based processors that execute network functionality.

5.1 Programming Abstraction

As a primary programming abstraction, we draw inspiration from stream processing systems like Dataflow SQL [15] and view each RPC as a tuple with one or more fields. Elements process an incoming stream of tuples, and their processing logic is specified in a SQL-like DSL, which is then compiled to native device code. Each element can read or write internal states modeled as tables. The processing logic outputs zero or more tuples. When an RPC is modified, the output fields differ from the input fields. Downstream elements in the pipeline can read and further edit these fields.

Figure 4 shows an element that implements access control. The element holds its state in the `ac_tab` table that stores a mapping between `username` to `permission`. It uses this state to generate an output table based on the `input` table, which has a single row with incoming RPC. The element blocks users who do not have write permissions and is executed upon every RPC arrival, sending new outputs downstream.

A SQL-like language provides a foundation for the compiler to infer which fields are read or written by an element, when it is safe to re-order elements, and what information needs to be communicated between elements (headers). We can build on it to enable additional capabilities that we need.

However, SQL cannot express certain forms of complex processing that we need. One such class is operations like compression and encryption. We can model these as user-defined functions for which developers provide platform-specific implementations. This approach is similar to how Tensorflow [22] requires platform-specific implementation of complex operators. Another class of complex processing involves "shaping" the RPC stream via mechanisms such as timeouts, retries, and congestion control. We can introduce special elements of type filters to express their operation. Simple filters will be expressed in (extended) SQL, and complex ones will use operators with platform-specific implementations. Depending on application needs, these operators may even wrap around an existing protocol such as TCP.

5.2 Control Plane

The ADN controller is a logically centralized component that has global knowledge (acquired via cluster managers such as Kubernetes [10]) of the network topology, service locations, and available ADN processors. It provisions network processing on available processors.

In response to workload changes and failures, it also migrates and scales ADN elements. The decoupling of code and state, and the tabular nature of state, enables us to reconfigure the network without disrupting applications. To migrate or scale out a load balancer, the controller can copy over its state and start running a new instance; while reducing the number of load balancer instances, it can merge their states and kill some instances. Some reconfigurations may require us to put the network in intermediate states to prevent transient disruptions [35, 48, 55]. State decoupling also enables us to hot-update element processing logic [34].

The ADN compiler takes ADN elements (defined in our SQL-like DSL) and generates efficient implementations for the target platform. Internally, the compiler first converts the program into an intermediate representation (IR). It then applies a set of optimizations on the IR. For example, if two elements do not operate on the same RPC fields, they can be executed in parallel. Finally, the compiler translates optimized IR into platform-native code.

5.3 Data Plane

The ADN data plane is composed of ADN processors that carry out the low-level executions of ADN elements. Each processor acquires the compiled version of the RPC processing logic from the control plane and periodically sends reports of logging, tracing, and runtime statistical information back to the controller. ADN processors can be realized in either software (for example, in the form of RPC libraries, user-space proxy, or eBPF) or hardware (such as SmartNIC or programmable switches). An ADN processor might only manage a portion of a processing graph, and if that's the case, the RPC headers might convey additional information intended for the utilization of downstream processors.

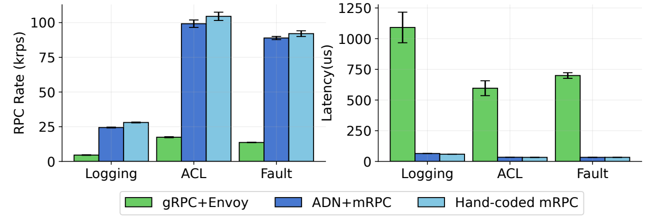

Figure 5: Performance of ADN compared to Envoy and hand-coded mRPC modules.

6 Preliminary Prototype and Evaluation

To help evaluate the feasibility and performance of the approach above, we implemented a prototype. Our prototype integrates with Kubernetes. We created a Kubernetes custom resource [20] called ADNConfig which developers use to provide ADN programs. The ADN controller watches for changes to this resource or to the deployment (e.g., a new service replica). It updates the data plane processors when either changes. Our prototype only supports the mRPC [25], a managed RPC system service, as the processor. We use TCP/IP as the transport for mRPC. The ADN compiler converts the high-level SQL-based DSL to Rust-based mRPC modules (i.e., implementation of engines). Work on supporting other platforms is ongoing.

We implemented several elements in our DSL. The ones we use in our evaluation are: 1) Logging, which records both the request and response to a file, 2) Access Control List (ACL), which inspects RPC arguments and drops RPCs based on a set of rules, and 3) Fault Injection, which aborts requests based on a configured probability. Interestingly, standard SQL syntax was rich enough for these elements.

Experimental Setup. We evaluate our prototype using a simple client-server application. The client keeps sending 128 concurrent RPC requests using a single thread. Both the RPC request and response contain a short byte string. The ADN network specification chains the three elements mentioned above. That is, RPCs are logged, access controlled, and some of them are dropped.

We compare the performance of the prototype to the standard approach of using Envoy proxy with gRPC. We also compare against hand-written mRPC modules to understand the ease of development in our DSL versus Rust (the language of mRPC) and the performance impact of auto-generated code. The mRPC modules were written by mRPC developers for high performance. We run the experiments on two machines with two Intel 10-core Xeon Gold 5215 CPUs and 256 GB RAM, Ubuntu 20.04. We use Envoy v1.20.

Results. Figure 5 shows the results. ADN provides a 5–6x higher RPC rate and 17–20x lower RPC latency compared to using Envoy for the same network functionality. The performance overheads of using Envoy is from the current service mesh architecture, which needs to parse/serialize standard protocol (gRPC, HTTP) headers and has extra marshalling/unmarshalling of RPC payloads. Envoy's RPC processing is also more expensive because the filters for logging, access control, and fault injection are more general with more knobs than our application needs.

Compared to hand-optimized mRPC modules, ADN modules have 3–12% lower performance. This degradation is primarily due to the programming abstraction of ADN. In terms of lines of code, a proxy for ease of development, ADN elements have tens of lines of SQL, whereas hand-written mRPC modules have hundreds of lines of Rust.

7 Discussion

Do ADNs require application source code modification? No. We can realize ADNs without source code modification by modifying RPC libraries like gRPC. Applications send and receive RPC messages via such libraries, and our modifications will process messages and forward them to other processors based on the implementation determined by the ADN controller. By linking against the modified library, ADNs can be a drop-in replacement for existing service meshes [19, 21].

How do ADN applications communicate externally? ADN focuses on building a network tailored to an application but this application may need to communicate with other applications and external clients. As with service meshes, such communication can happen via designated ingress and egress locations for an application. The ingress locations translate incoming IP packets into the ADN format, and the egress locations do the reverse translation.

When two ADN-based applications communicate, instead of translating the sender ADN's messages to a standard format and then translating the standard format to the receiver ADN's format, we can directly translate information between the two ADNs. Such "application peering" not only removes one translation step but also eliminates the need to "downshift" application messages to IP and back.

Are there other domains where the ADN approach applies? Yes. There are domains beyond microservices where custom communication functionality is needed to support distributed application endpoints, including data analytics [43, 61], and distributed ML training [41, 58]. These domains can also benefit from the ADN approach of auto-generating a network implementation based on a high-level specification. We do expect, however, that the specification language for different contexts will be different to accommodate the unique needs of each domain.

8 Related Work

Reducing the overhead of application networks. The overhead of application networks (e.g., service meshes) is well-recognized and there are ongoing (not productized yet) efforts to lower them [5, 8]. These approaches aim to lower the performance overheads of application networks for some types of processing but they fall back to sidecars in the general case. They also still follow the same set of standardized network abstractions. ADN is a fundamentally different approach: remove (almost) all standard network abstractions and create a fully custom network implementation.

ServiceRouter [57] lowers the overhead of application networks for a specific functionality (request routing), by supporting multiple deployment modalities based on the service and its environment. ADN will enable this capability for diverse, developer-specified network functionalities.

Application-specific network customization. Clark and Tennenhouse articulated the shortcomings of fixed network layers over three decades ago and proposed Application Level Framing (ALF) [27] for packets. Others too have made similar observations and explored alternatives [26, 33, 40] where headers and some network functionality are customized to the applications. We take this perspective further. Instead of looking at the network as a communication substrate for layer-3/4 connectivity, we view the "network" in application networks as including all the rich processing (e.g., load balancing, telemetry, access control). We also argue for high-level specifications and auto-generated implementations.

High-level network programming. There is a rich line of work on specifying aspects of network behavior in a higher-level language and automatically generating low-level implementations. Declarative Networking [46] uses Datalog to express layer-3 control plane protocols such as OSPF; NetKat [23] and similar languages [31, 36, 54] express end-to-end packet forwarding based on layer 2-4 headers; and Rubik [44] expresses middlebox processing of IP packets. We draw inspiration from these works, but our target domain is different—application-specific abstractions and message processing, without relying on the existing layered model.

9 Conclusion

With ADN, developers specify the network functionality desired by the application in a high-level language. A distributed implementation that is customized to the application and deployment environment is then automatically generated. ADNs not only fit the application like a glove—they have all the functionality that the application needs and nothing more—but they can also leverage heterogeneous hardware and scale with the workload.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful feedback. This work is supported by NSF (grants CNS-2238665, FMiTF-2219863), UW FOCI and its partners (Alibaba, Amazon, Cisco, Google, Microsoft, and VMware).

References

[1] 2003. RTP: A Transport Protocol for Real-Time Applications. https://www.rfc-editor.org/rfc/rfc3550.
[2] 2007. Stream Control Transmission Protocoltrp. https://www.rfc-editor.org/rfc/rfc4960.html.
[3] 2021. Benchmarking Linkerd and Istio: 2021 Redux. https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/.
[4] 2022. Building simplified service mesh API for developers. https://events.istio.io/istiocon-2022/sessions/building-simplified-service-mesh-api-for-developers/.
[5] 2022. Cilium Service Mesh – Everything You Need to Know. https://isovalent.com/blog/post/cilium-service-mesh/.
[6] 2022. gRPC Proxyless Service Mesh. https://istio.io/latest/blog/2021/proxyless-grpc/.
[7] 2022. HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/.
[8] 2022. Introducing Ambient Mesh. https://istio.io/latest/blog/2022/introducing-ambient-mesh/.
[9] 2022. Istio: Performance and Scalability. https://istio.io/latest/docs/ops/deployment/performance-and-scalability/.
[10] 2022. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.
[11] 2022. NGINX: Advanced Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/.
[12] 2022. Performance Impacts of an Istio Service Mesh. https://pklinker.medium.com/performance-impacts-of-an-istio-service-mesh-63957a0000b.
[13] 2022. Service meshes are on the rise — but greater understanding and experience are required. https://www.cncf.io/wp-content/uploads/2022/05/CNCF_Service_Mesh_MicroSurvey_Final.pdf.
[14] 2022. Taming Istio Configuration with Helm. https://events.istio.io/istiocon-2021/sessions/taming-istio-configuration-with-helm/.
[15] 2022. Use Dataflow SQL. https://cloud.google.com/dataflow/docs/guides/sql/dataflow-sql-intro.
[16] 2022. What is network virtualization? https://www.vmware.com/topics/glossary/content/network-virtualization.html.
[17] 2023. Amazon Virtual Private Cloud (Amazon VPC). https://aws.amazon.com/vpc//.
[18] 2023. Envoy. https://www.envoyproxy.io/.
[19] 2023. The Istio service mesh. https://istio.io/.
[20] 2023. Kubernetes Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.
[21] 2023. The world's lightest, fastest service mesh. https://linkerd.io/.
[22] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: a system for large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.
[23] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. *Acm sigplan notices* 49, 1 (2014), 113–126.
[24] Sachin Ashok, P Brighten Godfrey, and Radhika Mittal. 2021. Leveraging Service Meshes as a New Network Layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. 229–236.
[25] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. 2023. Remote Procedure Call as a Managed System Service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 141–159.
[26] Isabelle Chrisment, Delphine Kaplan, and Christophe Diot. 1998. An alf communication architecture: Design and automated implementation. *IEEE Journal on Selected Areas in Communications* 16, 3 (1998), 332–344.
[27] David D Clark and David L Tennenhouse. 1990. Architectural considerations for a new generation of protocols. *ACM SIGCOMM Computer Communication Review* 20, 4 (1990), 200–208.
[28] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. 2021. Offloading load balancers onto SmartNICs. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. 56–62.
[29] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering* (2017), 195–216.
[30] 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.
[31] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A network programming language. *ACM SIGPLAN Notices* 46, 9 (2011), 279–291.
[32] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling innovation in network function control. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 163–174.
[33] Dongsu Han, Ashok Anand, Fahad Dogar, Boyan Li, Hyeontaek Lim, Michel Machado, Arvind Mukundan, Wenfei Wu, Aditya Akella, David G Andersen, et al. 2012. XIA: Efficient support for evolvable internetworking. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 309–322.
[34] Michael Hicks, Jonathan T Moore, and Scott Nettles. 2001. Dynamic software updating. *ACM SIGPLAN Notices* 36, 5 (2001), 13–23.
[35] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic scheduling of network updates. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 539–550.
[36] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. 2015. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 59–72.
[37] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 863–880.
[38] Eddie Kohler, Mark Handley, and Sally Floyd. 2006. Designing DCCP: Congestion control without reliability. *ACM SIGCOMM Computer Communication Review* 36, 4 (2006), 27–38.
[39] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.
[40] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. 2007. A data-oriented (and beyond) network architecture. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*. 181–192.
[41] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked*

Systems Design and Implementation (NSDI 21). 741–761.

[42] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: Efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 36–51.

[43] Alberto Lerner, Rana Hussein, Philippe Cudre-Mauroux, and U eXascale Infolab. 2019. The Case for Network Accelerated Query Processing.. In *9th Biennial Conference on Innovative Data Systems Research (CIDR 19)*.

[44] Hao Li, Changhao Wu, Guangda Sun, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. 2021. Programming Network Stack for Middleboxes with Rubik. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 551–570.

[45] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 387–406.

[46] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative networking. *Commun. ACM* 52, 11 (2009), 87–95.

[47] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*. 412–426.

[48] Ratul Mahajan and Roger Wattenhofer. 2013. On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. 1–7.

[49] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating complex network services with eBPF: Experience and lessons learned. In *IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 1–8.

[50] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating network applications with stateful TCP offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 77–92.

[51] Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. 2021. Cerebros: Evading the RPC tax in datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 407–420.

[52] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. 2022. SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 780–794.

[53] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 772–787.

[54] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. 2013. Modular SDN programming with pyretic. *Technical Report of USENIX* 30 (2013).

[55] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 323–334.

[56] Jerome H Saltzer, David P Reed, and David D Clark. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)* 2, 4 (1984), 277–288.

[57] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/osdi23/presentation/saokar

[58] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. 2021. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 785–808.

[59] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 23)*. 498–514.

[60] Brent E Stephens, Darius Grassi, Hamidreza Almasi, Tao Ji, Balajee Vamanan, and Aditya Akella. 2021. TCP is Harmful to In-Network Computing: Designing a Message Transport Protocol (MTP). In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. 61–68.

[61] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2407–2422.

[62] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. 2020. Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[63] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 2020. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the ACM SIGCOMM 2020 Conference*. 283–295.

[64] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. 2022. Aequitas: Admission control for performance-critical rpcs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 1–18.

[65] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 655–672.

[66] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. 2023. Dissecting Service Mesh Overheads. In *Proceedings of the ACM Symposium on Cloud Computing (to appear)*.