Application Defined Networks

Xiangfeng Zhu Weixin Deng Tianyi Cui
Thomas Anderson Arvind Krishnamurthy Ratul Mahajan Danyang Zhuo

UW FOCI (https://foci.uw.edu/)

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. 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.

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 [26]. 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 [2, 3, 29, 38, 58]. For instance, many distributed applications need load balancing across replicas, something that the Internet does not provide.

High overhead and imperfect application support maybe inevitable for a general network, but many networks today are built to support a single application. The key driver for such application networks are microservices [28], where application logic is split across many (sometimes thousands [49, 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 [20] 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 or development at 90% of organizations that develop cloud applications [15].

The tragedy of today's application networks is that, even though they serve a single application, they are built using the same abstractions that were 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 architecture, which allows service meshes to support a range of applications, has significant downsides. Depending on service mesh configuration, it can increase message processing latency by up to 2.7-7.1x and CPU usage by up to 1.6-7x [4, 12, 14, 54, 67]. Further, layering hides or obscures information, which makes it hard to implement network policies that are highly application specific (e.g., choosing replicas based on information in the application's RPC) [5, 16]. Finally, being general, service mesh implementations are large and complex, so it is almost impossible to accelerate them via programmable kernel, NICs, and switches [27, 51, 52, 60].

Based on these observations, we argue for application networks to be highly customized to the application and its deployment environment. This perspective is the natural endgame for the lines of work that adapt standard protocols to better meet the application needs [1, 10] and do cross-layer optimizations to lower the overhead of generality [32, 41, 47].

The key design challenge is: can we 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 an underlay network that only provides 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, 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 [58]. 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 up without disruption, as the number of microservice instances changes or the workload scales.

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 sharded 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 (e.g., we may introduce a third replica for B), 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. The application may or may not have cared about these layers originally.

Then, the developer selects a module that can enforce their policies; this functionality is common in L7 proxies [7, 9, 13].

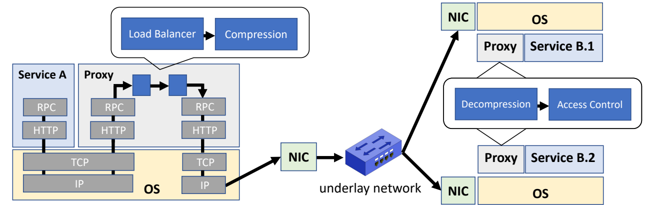

Figure 1: Packet processing in service meshes.

Finally, they need a mechanism such that 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.

The resulting packet path and processing are shown in Figure 1. 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 [20, 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 its downsides are significant.

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 [54]. This overhead is on top of the already-expensive communication cost in microservices even without service meshes [43, 53]. The overhead of service meshes stems from parsing all the protocol headers to recover wrapped information [67]. 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 [51, 62] or hardware [27, 52, 60].

As just one difficulty, using programmable networking hardware often requires custom header designs due to hardware constraints [42, 46]. 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 service meshes would 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 them hard to develop [5, 16] and the isolation mechanisms for safely running these plugins (e.g., Web Assembly) further drive up the overhead [67].

3 Application defined networks

Given the pitfalls of building application networks using general-purpose abstractions, we advocate building them in a way that is highly customized to the application and its 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 [19]) provide this abstraction, and technologies like VXLAN [18] can implement it anywhere.

Everything the network does beyond layer 2 is specified by the application developer in a domain-specific language (DSL). This specification includes a chain of *elements*, each is an operation on an RPC message between two services. A controller decides how to realize the specification in the deployment environment of the application. 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, or on a programmable hardware device, or a mix of locations. The controller may also decides to execute multiple elements in parallel or re-order them for efficiency.

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 [8]). 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

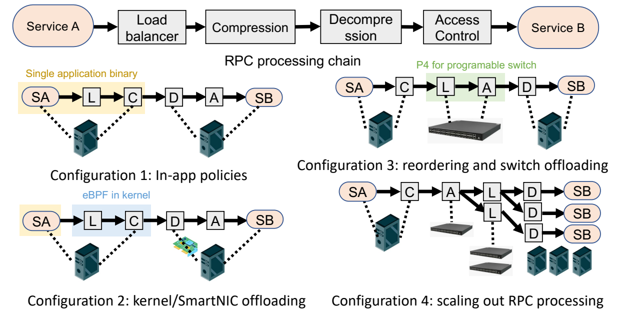

Figure 2: Possible realizations of a RPC processing chain (described in §2).

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 is also automatically determined.

3.1 Key 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 internal state of elements because that is key for seamless migration and scaling [31].

Specifying network processing using chained elements is not new. A seminal system is Click [39] which enables building modular packet processing pipelines. Packet processing can be 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 [31, 55].

We also want to enable developers to reuse code of elements developers 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

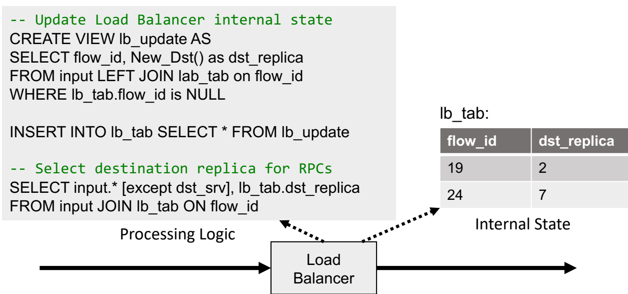

Figure 3: An element that does sticky load balancing.

reliability constraints and any element location constraints (e.g., encryption element must be co-located with the sender and not offloaded to a network switch).

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 for kernel, P4 for programmable switches) 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.

3.2 Potential Approach

We do not have definitive answers to the questions above; we outline our preliminary thinking. Drawing inspiration from stream processing systems like Dataflow SQL [17], we 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 whose fields may differ from those of the input tuples (when an RPC is modified). Downstream elements in the processing pipeline can read and edit these fields. Using a SQL-like language enables the compiler to infer which fields are read/written by each element, when it is safe to re-order elements, and what information needs to be communicated between elements (headers).

Figure 3 shows an example element that does sticky load balancing. The element holds its state in the `lb_tab` table that stores a mapping from `flow_id` to `dst_replica`. It uses this state to generate an output table based on the `input` table which has a single row with incoming RPC. The element first looks at the `input` to see whether there is an RPC row for which `flow_id` is not in `lb_tab`. If so, there is no sticky requirement for that RPC and it calls the `New_Dst()` function to randomly pick a destination replica and store the decision in `lb_tab`. The second part of the processing logic generates an output table that contains the load balancing decision. The element's code is run upon every RPC arrival and new outputs are sent downstream.

We can build on this basic language abstraction to provide many of the capabilities we need.

Reusing elements. How can we reuse the code of our load balancer above, which expects a field called `flow_id`, to process application RPCs that may not have such a field? We use the idea of SQL table views to enable such reuse. Elements are seen as operating on table views, and when importing them for use by an application, developers provide a view definition query that maps input RPC table to the element's view. An application that wants to use its RPC's `object_id` field for load balancing can provide this query: `SELECT *, object_id AS flow_id FROM input`. With such reuse, most developers do not need to develop their own element code. Specifying RPC processing will be similar to chaining filters in service meshes today.

Supporting complex processing. SQL cannot (easily) 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 special introduce 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.

Cross-element optimization. Since each element is represented in a SQL-like language, it is possible for cross-element optimization leveraging SQL query rewriting and plan generation techniques. Other system-level optimizations are also possible. When multiple elements all have states, they can represent those states into an aggregated table state to facilitate efficient table lookup. Different elements can also work on a single copy of the RPC in shared memory.

Computing performant ADN configurations. Depending on available resources in the application's environment, there are multiple ways to realize RPC processing (Figure 2). To calculate the optimal runtime configuration, we first need a way to compute the expected resource consumption for different ADN configurations. One solution is to use a model-based approach [67]. Using an offline profiling process, we will first build a profile for each processing element on each platform where it can run. An element's profile includes its performance and resource usage metrics. Based on these profiles, we can estimate end-to-end performance and resource usage based on a compositional model. After that, we can formulate a search space for the configuration and use an optimization method to find optimal configurations.

Disruption-free reconfiguration. In response to workload changes and failures, ADN elements will be migrated and scaled up/down. 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, we can copy over its state and start running a new instance; while scaling down load balancer instances, we 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, 50, 57]. State decoupling also enables us to hot-update element processing logic, using dynamic software updates [34].

4 Discussion

This section discusses a few questions relevant to ADNs.

Do ADNs require application source code modification? Not necessarily. We can realize ADNs without source code modification for a large class of applications by modifying RPC libraries like gRPC. Applications send and receive RPC messages via such libraries, and our modifications will process and package messages according to the implementation determined by the ADN controller. By linking against the modified library, ADNs can be a drop-in replacement for existing service meshes [20, 21].

How do ADN-based 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.

Interesting opportunities arise when two ADN-based applications communicate. In that case, 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? There are other domains beyond microservices where custom communication functionality is developed to support multiple endpoints of the same application. These include in-network computation applications [66] such as data analytics [44, 61], and distributed ML training [42, 59]. These contexts can also benefit from the ADN approach of auto-generating a network implementation based on a high-level specification. We do expect the specification language for different contexts will be different to accommodate the needs of each domain.

5 Related Work

Reducing the overhead of application networks. The overhead of application-level networks (e.g., service meshes) is widely recognized in the industry and there are ongoing (not productized yet) efforts to lower them [6, 11]. All of these efforts can lower the performance overheads of application-level networks in certain conditions and they fall back to sidecars (or userspace proxies) in the general case. They still follow the same set of standardized network abstractions. We propose a fundamentally different approach of removing all the standardized network abstractions and thus expensive layered protocol parsing is not needed.

Application-specific network customization. Clark and Tennenhouse articulated the shortcomings of fixed network layers over three decades ago and proposed Application Level Framing (ALF) [26] for packets. Others too have made similar observations and explored alternatives [25, 33, 40] where headers and some network functionality are customized to the applications. However, our perspective differs from ALF and these proposals. They look at the network only as a communication substrate between endpoints, while today's application networks have a huge emphasis on in-network processing (e.g., load balancing, telemetry, access control). So, ADNs consider both the software on the endpoint and the in-network processing. In addition, we argue for considering the practical constraints of network hardware and drive network implementations via high-level specifications.

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 [48] uses Datalog to express layer-3 control plane protocols such as OSPF; NetKat [23] and similar languages [30, 36, 56] express end-to-end packet forwarding based on layer 2-4 headers; and Rubik [45] 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.

6 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.

References

[1] 1994. T/TCP – TCP Extensions for Transactions Functional Specification. https://www.rfc-editor.org/rfc/rfc1644.html.

[2] 2003. RTP: A Transport Protocol for Real-Time Applications. https://www.rfc-editor.org/rfc/rfc3550.

[3] 2007. Stream Control Transmission Protocoltrp. https://www.rfc-editor.org/rfc/rfc4960.html.

[4] 2021. Benchmarking Linkerd and Istio: 2021 Redux. https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/.

[5] 2022. Building simplified service mesh API for developers. https://events.istio.io/istiocon-2022/sessions/building-simplified-service-mesh-api-for-developers/.

[6] 2022. Cilium Service Mesh – Everything You Need to Know. https://isovalent.com/blog/post/cilium-service-mesh/.

[7] 2022. Envoy: Cloud-native high-performance edge/middle/service proxy. https://github.com/envoyproxy/envoy.

[8] 2022. gRPC Proxyless Service Mesh. https://istio.io/latest/blog/2021/proxyless-grpc/.

[9] 2022. HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/.

[10] 2022. HTTP/1.1: Chunked Transfer Coding. https://www.rfc-editor.org/rfc/rfc9112.

[11] 2022. Introducing Ambient Mesh. https://istio.io/latest/blog/2022/introducing-ambient-mesh/.

[12] 2022. Istio: Performance and Scalability. https://istio.io/latest/docs/ops/deployment/performance-and-scalability/.

[13] 2022. NGINX: Advanced Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/.

[14] 2022. Performance Impacts of an Istio Service Mesh. https://pklinker.medium.com/performance-impacts-of-an-istio-service-mesh-63957a0000b.

[15] 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.

[16] 2022. Taming Istio Configuration with Helm. https://events.istio.io/istiocon-2021/sessions/taming-istio-configuration-with-helm/.

[17] 2022. Use Dataflow SQL. https://cloud.google.com/dataflow/docs/guides/sql/dataflow-sql-intro.

[18] 2022. What is network virtualization? https://www.vmware.com/topics/glossary/content/network-virtualization.html.

[19] 2023. Amazon Virtual Private Cloud (Amazon VPC). https://aws.amazon.com/vpc//.

[20] 2023. The Istio service mesh. https://istio.io/.

[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 *Osdi*, Vol. 16. Savannah, GA, USA, 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] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[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] 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.

[42] 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.

[43] 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.

[44] Alberto Lerner, Rana Hussein, Philippe Cudre-Mauroux, and U eXascale Infolab. 2019. The Case for Network Accelerated Query Processing.. In *CIDR*.

[45] 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.

[46] 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.

[47] Xiaojun Lin, Ness B Shroff, and Rayadurgam Srikant. 2006. A tutorial on cross-layer optimization in wireless networks. *IEEE Journal on Selected areas in Communications* 24, 8 (2006), 1452–1463.

[48] 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.

[49] 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.

[50] 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.

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

[52] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading.. In *NSDI*, Vol. 20. 77–92.

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2021. Scaling distributed machine learning with in-network aggregation. In *NSDI*, Vol. 21. 785–808.

[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 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 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 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*.

[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] Bohan Zhao, Wenfei Wu, and Wei Wu. 2023. NetRPC: Enabling In-Network Computation in Remote Procedure Calls. In *NSDI*, Vol. 23.

[67] 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. 2022. Dissecting Service Mesh Overheads. *arXiv preprint arXiv:2207.00592* (2022).