Full-stack SDN: The Next Big Challenge?

Gianni Antichi
Queen Mary University of London
g.antichi@qmul.ac.uk

Gábor Rétvári
MTA-BME Information Systems Research Group
Ericsson Research, Hungary
retvari@tmit.bme.hu

ABSTRACT

This paper challenges the common assumption that SDN networks shall be run only at lowest layers of the stack, i.e., L2 and L3. Using as use case data center networks providing virtualized services, we show how state-of-the-art solutions already employ some application-level processing via a central controller. With this in mind, we question if the lessons learned from a decade of SDN networking can be also extended to the upper layers. We make the case for a full-stack SDN framework that encompasses all protocol layers in the network stack, and call for further research in the area.

CCS CONCEPTS

• **Networks** → **Layering**; **Programming interfaces**.

KEYWORDS

Software Defined Networks, service mesh, network architectures

ACM Reference Format:
Gianni Antichi and Gábor Rétvári. 2020. Full-stack SDN: The Next Big Challenge?. In *Symposium on SDN Research (SOSR '20), March 3, 2020, San Jose, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3373360.3380834

1 INTRODUCTION

Are Service Meshes the Next-gen SDN?
DevOps Zone, 2017

Layer-7 is the New Layer-4
Cilium, future:net, 2017

Data center networks are hard to manage [22]. This is due to their intrinsic complexity: Microsoft reported that just a single site can be composed by hundreds of thousands of servers and switches, combined across multiple components through millions of cables and fibers [12]. The rise of end-host virtualization, consolidating possibly a very large number of diverse services on a single system, and the rapid convergence of cloud-native service access APIs based on the HTTP communication protocol [26, 35], has further increased the pressure on network operators with a new layer of complexity.

In response to the above concerns, a dedicated communication overlay (the *service mesh* [8, 15, 28]) has been recently proposed as a potential technology to manage network communications at the application layer (Layer-7, or L7). In this architecture (see Figure 1), services or applications[1] are generally connected to a *sidecar proxy*, which is a L4–L7 design pattern abstracting features, such as inter-service communications, monitoring and security, to ease the tracking and maintenance of an application as a whole. The sidecar proxy is then connected to a *virtual switch* (or vswitch) that acts as a L2-L3 gateway to the physical network infrastructure. Applications running on the same server interact only via the sidecar proxy and the virtual switch, while services located in two different hosts need the data center fabric to exchange traffic, through the respective proxies and virtual switches [22, 49].

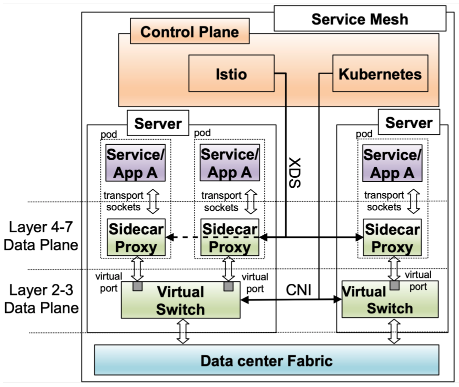

Figure 1: An L7–SDN: The Service Mesh.

Interestingly, the last 15 years of fast-pace evolution has resulted in a somewhat haphazard separation of concerns in data center networking: the fabric is managed by a cloud orchestration framework like OpenStack [48], Andromeda [4], or AccelNet [7], basic node-level L3 network virtualization is provided by a container orchestration system like Kubernetes [16], Mesos [13], or Docker, on top of which a service mesh framework, like Istio [15], linkerd [8], or Consul, delivers enhanced L7 network virtualization [28]. This "split-identity" network virtualization approach has obtained the opposite result of the original proposition: a skyrocket complexity in network management [22] which has introduced new security concerns [49, 50] and performance bottlenecks [8, 11].

In this paper, we ask whether a *unified, full-stack approach* may help alleviate mounting data center network management issues. Our premise is the observation that, similarly to data center virtual networking [22], the service mesh follows a design pattern familiar to the SDN principles: a central authority (e.g., Istio [15]) that exerts fine-grained control over global L7 network policies by programming the underlying L7 data plane (the sidecar proxies). The difference is that the service mesh operates at higher layers in

[1]From now on, we will use those two terms interchangeably.

the protocol stack, while typical SDN deployments operate at L2 and L3, and partially at L4.

We thus wonder if the lessons learned from a decade of SDN networking can be also extended into the service mesh model, and to application-layer network virtualization as a whole. This is challenging because the required API abstractions for higher layers change much faster than at lower layers: applications' needs rapidly change while IP and transport protocols are less frequently subject to complete redesigns.

We make the case for a full-stack SDN framework that encompasses practically all protocol layers in the network stack, from L2 to L7, and call for further research in the area. We consider two prominent data-center networking use cases, the state-of-the-art application of the service mesh to provision Web services and a proof-of-concept L7 Network Function Virtualization (NFV) prototype. The operational experience we obtain is then leveraged to sketch a research program towards developing a full-stack SDN framework.

The main contributions of the paper are as follows:

- We position the service mesh as an *L7–SDN* by identifying the sidecar proxy as the programmable L4–L7 data-plane and Istio as the control plane.
- We consider two L7–SDN use cases to obtain operational experience and we identify critical architectural concerns regarding the current state-of-the-art.
- We sketch a research program towards the development and associated challenges for a full-stack SDN solution.

2 THE CASE FOR FULL-STACK NETWORKING

In this section we first discuss the need for insights into higher layers of the protocol stack and draft the requirements for a generic L7-aware data center network infrastructure.

The need for HTTP-aware insight. Traditionally, TCP/IP provided a minimal but sound elementary end-to-end network experience to applications. The increasing generality of higher-layer protocols, such as HTTP, has led operators to explore their feasibility beyond the Web, i.e., for media streaming (WebRTC), remote-procedure calls (gRPC), and data center networking [18]. Slowly but steadily, this has made HTTP the new "narrow waist" in the IT protocol stack [35]. Crucially, when everything is encoded into HTTP then basic L2–L4-layer insight into traffic is no longer adequate to exert full control over the network [26]. A "traditional" firewall filters only on the "IP 5-tuple" (IP source and destination addresses, transport protocol and ports), but this is of little use when the destination port is uniformly 80 (HTTP) or 443 (HTTPS) regardless of whether a particular traffic instance is a REST API call or a long-lived media stream.

L2–L7 networking beyond HTTP. Beyond the Web application space, wireline and mobile telecommunications, NFV, game networking, industrial automation, and finance still rely on use-case-specific legacy L7 protocols. Similarly to HTTP-bound Web services, these areas would likewise benefit from managing, load-balancing, filtering and monitoring traffic at the granularity of individual L4–L7 sessions. This requires generic application-specific insight into network communications. At the same time, L2–L3 networking remains indispensable in the foreseeable future, for core data-center networking and low-level network virtualization. In summary, a modern SDN framework should be able to exert fine-grained programatic control over *any* layer in the protocol stack in a protocol-agnostic manner (not bound to HTTP).

Table 1: Feature comparison between standard telco-grade SDN/NFV frameworks and the service mesh.

Feature/Pattern (related NFV literature)	OSM + OpenStack	ONAP + OPNFV	NSM [31]	K8s/Istio
Declarative traffic management [19, 33, 47]	✓	✓	✓	✓
Service discovery	✓	✓	✓	✓
Service gateways [44]	✓	✓	×	✓
Monitoring	✓	✓	C	✓
Adaptive load-balancing [19, 33, 40, 47]	T	T	C	✓
Transparent encryption [34, 37]	×	×	×	✓
Canary, A/B testing, CI/CD	×	×	×	✓
Fault injection, tracing [29]	×	×	C	✓
Health-check, timeout, retries, circuit breaking [23]	T	T	C	✓
Public cloud deployability	✓	✓	×	✓
Development activity (all time devs/commits as of 2019/08/01)*	high 118/4026	high 173/2965	N/A	very high 457/10126
User on-boarding, ease of use**	moderate	hard	moderate	very easy
QoS (throughput/latency) [51]	✓	✓	C	×
Telco standards compliance	✓	✓	×	×
Carrier-grade performance (packet throughput/latency)	✓	✓	C	×

✓: supported, C: claimed/proof-of-concept, T:, third-party/externally provided, ×: not supported/provided
*: based on Open Hub [32] **: according to own experience

SDN features at L7. Table 1 compares the features available in traditional NFV/SDN frameworks[2] against Istio, taken as a representative control plane for a service mesh architecture. We see several overlapping features but, as we point out below, these features are provided at different layers of the protocol stack, which bears critical architectural consequences. For instance, both de facto SDN frameworks and Istio enable rule-based *traffic management*, but whereas in the former this is possible only at the granularity of the IP 5-tuple, Istio can also route individual HTTP calls separately depending on the targeted endpoint or request cookies [15] and apply load-balancing policies on a per-application basis [27]. Similarly, *security and policy enforcement* and *tracing, monitoring, and logging* occurs at a much coarser grain in de facto SDN, whereas for Istio L7-aware protocol insight allows to filter traffic even at the granularity of individual REST API endpoints, selectively encrypt/decrypt privacy-sensitive application streams, and monitor network operations beyond traditional L2–L4 statistics from live data-plane traces [10, 20]. On top of these, Istio provides a unique set of L7-specific features: *timeouts/retries and circuit breaking* allow Istio to handle failures gracefully by automatically retrying and timing out failed L4–L7 connection attempts and removing unavailable service endpoints from the load-balancing pools (circuit breaking), and Istio's support for canary rollouts, A/B testing, "dark launch" [42], staged rollouts with percentage-based traffic splits, and chaos testing through programmable failure injection unlocks the best *continuous integration/delivery practices* (CI/CD) which are currently unparalleled in the de facto SDN world.

[2]From now on, we refer to these as *de facto SDN*, to highlight that while SDN in principle is oblivious to the specific layer at which the packet processing is enforced, current deployments work mostly at L2 and L3.

The challenge: Full-stack SDN. At the moment, the L7-aware features of Istio are available only for HTTP-based Web services and, in a severely restricted form, for plain L4 TCP streams. At the same time, the underlying cloud orchestration platform acts as central control plane for basic L2–L3 network virtualization, providing many of the same features at lower layers of the protocol stack. In addition, possibly another control plane may be in charge of managing the underlying data center fabric. This creates a *split-stack SDN* architecture, where multiple control planes and data planes interact in complex ways to deliver a full-stack L2–L7 network experience to applications.

This paper challenges current operational practices in data center networking and questions if the lessons learned from a decade of SDN networking can be also extended to higher layers in the protocol stack. In particular, we ask:

(1) Can we extend SDN best-practices from the current L2-L3–SDN model to higher protocol layers, up to L7?
(2) Can we build a unified full-stack SDN, spanning L2–L7 in the protocol stack, to alleviate mounting network management issues?

3 THE SERVICE MESH AS AN L7–SDN

Starting from Figure 1 as a reference, in this section we investigate the similarities between the current service mesh architecture and the de facto SDN approach.

The Istio architecture. Originally, different subsets of the service mesh feature set existed scattered throughout piecemeal implementations, software libraries, and SDKs, which made it difficult to reuse these features across different programming languages and development frameworks, and to centrally impose global communication policies. This has led to the development of the programmable L4–L7 *service proxy*, a language and framework agnostic tool to apply advanced application-layer communication policies on network traffic. The service proxy acts as a single point of policy enforcement and network service abstraction. The policies themselves are programmed into proxies by a *centralized control plane* in concert with the high-level intents defined by the mesh operator. The resultant service mesh architecture (recall Fig. 1) then lends itself readily to be casted in the framework of L7–SDN.

Data plane: Envoy, a programmable L4–L7 proxy. Envoy is an open-source L4–L7 proxy that can act as a load-balancer, a service gateway or a service proxy, depending on the deployment model. In the application-layer SDN framework, Envoy acts as a L7 counterpart of the "de facto SDN" switch: it receives incoming traffic through "L7 ports" (Listeners), e.g., on a server-side WebSocket or plain TCP receiver socket; processes ingress traffic using the familiar match-action abstraction (Filters), e.g., to load-balance on session cookies, rewrite headers, or to route requests per REST API endpoints; and forwards requests to upstream services (Clusters) and service backends (EndPoints) through client-side sockets. The individual components are loaded dynamically via a northbound API (xDS). The high-level architecture of Envoy is depicted in Fig. 2.

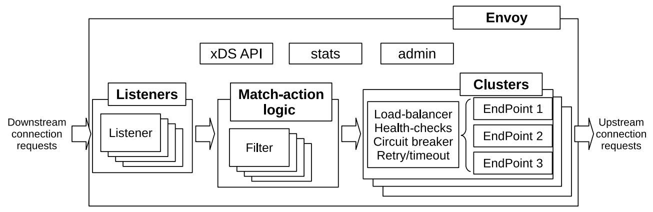

Figure 2: The programmable L7 data plane: Envoy.

Control-plane: Istio. The Istio control plane accepts a high-level, human-readable configuration describing global communication policies, synthesizes a configuration for each service proxy and applies it through xDS (`istio-pilot`). Furthermore, Istio contains a component to securely distribute secrets (`istio-citadel`) and another one to extract monitoring information from the service proxies and report these to a central store (`istio-mixer`).

The "Universal Dataplane API", xDS. Envoy accepts dynamic configuration from the control plane through the xDS API. Of particular importance here is the Listener Discovery Service (LDS), an API to manage ingress sockets, the Route Discovery Service (RDS) to determine the route for incoming session requests by matching on connection metadata, and the Cluster Discovery API (CDS) to dynamically configure the backends for each service. The collection of all these APIs is called the "Universal Dataplane API" [21], even though, as we argue later, it may not be *that* universal after all.

In summary, we see that Istio nicely fits into the SDN paradigm, with the control plane and the data plane cleanly separated and the former dynamically configuring the latter via an open API. Curiously even the data-plane architectures, from a mile high view, are similar, in that both the "de facto SDN" programmable switches and L7 service proxies are built on the venerable *match-action abstraction* [36], just the metadata matched differ (per-packet L2-L3 metadata versus per-connection HTTP header fields).

4 USE CASES

In this section, we take a closer look at two prominent data-center networking use cases and we argue that the state-of-the-art "split-stack" SDN approach incurs considerable management complexity. First, we point out that the default service mesh deployment model, where the vswitch and the service proxy are deployed separately, raises concerns even for the service mesh killer application, Web services. To gather further operational experience on an especially network-heavy use case, we also built an NFV prototype on top Istio; we summarize the takeaways and briefly report on the performance.

4.1 Web Services

There is a rising trend to provision cloud-based Web applications following the *micro-service model*, whereby the application is structured as a collection of loosely coupled distributed services that communicate over the network to collectively deliver the compound business logic. Cloud-native micro-services are usually implemented as Linux containers or *pods* and use technology-agnostic protocols, like HTTP, to serve and consume RESTful APIs. In the

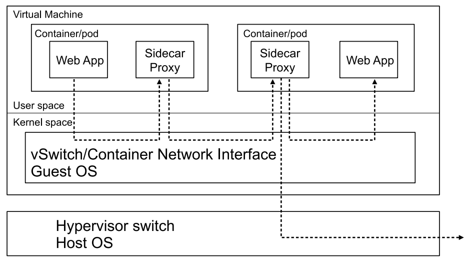

Figure 3: The sidecar proxy pattern.

	NFV/SDN	Service Mesh
Orchestration	ONAP, OP-NFV, OpenNetVM	Kubernetes + Istio
Protocol Layer	L2/L3/L4 + NSH	L7
Transport	GRE/MPLS/VxLAN + vSwitch	HTTP, gRPC, Websocket + Envoy
Communication Pattern	Long-lived connections, proxy mode (passthrough)	Short-lived request-response traffic (server mode)
Service Routing/ Chaining	Network Service Header (NSH) + ToR/vswitch routing	HTTP Headers/Cookies + Istio HTTP routing
QoS	Per-flow	Per-request
Language/Platform	dataplane: C/C++ + DPDK, ctrl plane: Java	dataplane: Node.js + OS network stack, ctrl plane: Go
Performance	Rate: 10–100 mpps, Latency: < 5 millisec/packet	Rate: 100+ Kreq/sec, Latency: 10+ millisec/req

Table 2: NFV/SDN vs. the service mesh.

micro-service model, the service mesh acts as a dedicated network infrastructure that automatically manages micro-service communication workloads across possibly hundreds of pod/container instances.

The sidecar proxy pattern. Figure 3 presents the state-of-the-art deployment model of Istio. In this model, basic L2–L3 network virtualization is provided by a virtual switch, or an equivalent policy routing pipeline (e.g., `iptables`) administered by the Container Network Interface (CNI) plugin, in the Guest OS kernel space. In order to enforce L4–L7 communication policies on top of this L2–L3 network virtualization layer, the service proxy (Envoy) needs to intercept all ingress and egress network connections to and from the application; this is ensured by *injecting* the service proxy as a sidecar alongside the application code and installing a complex policy routing chain in the pod namespace. We note that both the L2–L3 virtualization layer and the L4–L7 service proxies are managed by the same operator (the former via Kubernetes and the latter via Istio), yet the corresponding data planes are fully separated (but see [11] for an integrated approach). The sidecar proxy model has a number of critical consequences to service mesh operations, as we discuss next.

The Good. Deploying the service proxy in user space opens up the service mesh data plane to fast-pace innovation. Since the sidecar proxy is usually injected at application bootstrap time from an open container repository, new versions can be deployed rapidly and selectively for individual applications/services.

The Bad. Istio installs a fairly involved policy routing pipeline into the network namespace of each pod in order to transparently mediate ingress/egress communication from/to the application through the user-space sidecar proxy [46]. On the one hand, this enables tight control over application traffic, e.g., the mesh operator can close down TCP/HTTP access to all external services from an application. At the same time, the OS-based infrastructure to capture application traffic required by the sidecar proxy model introduces substantial extra complexity and latency/jitter. Currently, the capturing pipeline is restricted to TCP in Istio.

The Ugly. Container networking already takes a nontrivial tax on network performance [55], to which the sidecar proxy adds considerable extra overhead [8, 11]. In particular, in the user-space sidecar proxy model even a local packet exchange involves 3 separate phases (sender app to the sender sidecar proxy, between sidecars, and receiver sidecar proxy to the destination application) with 3 full round trips and 6 context switches between the two applications, the sidecar proxies, and the underlying vswitch/OS kernel (see again Fig. 3). For a remote packet exchange, the overhead is even larger. For comparison, if we were to substitute the sidecars and the vswitches with a *single* integrated L2–L7 SDN data plane component, then the overhead would reduce to 2 context switches and a single user-space to kernel-space roundtrip.

Takeaway. Even the killer service mesh application, Web services, struggle with certain complexities and performance penalties that arise from the current "split-stack" approach. A unified full-stack design would allow to sidestep some of these limitations.

4.2 Network Function Virtualization

After experimenting with Web services, where Istio truly shines, we turned to evaluate it with a traditional SDN-centric use case: we built a proof-of-concept NFV prototype on top of an unmodified Istio stack. In NFV, packet processing logic is decomposed into primitive network functions (NFs), each NF performing one single processing step on traffic traversing it, and the required packet processing logic is realized by dynamically chaining NFs one after the other. The NFV use case lends itself readily to be constructed on top of Istio, with NFs implemented as Kubernetes containers/pods and "NF plumbing" encoded as Istio routing rules.

Unfortunately, the target audiences for NFV and Istio are completely different; see a summary in Table 2. In order to bridge these gaps, we had to make a number of, sometimes rather taxing, design decisions.

Application-layer overlay. Istio exposes an application-layer network abstraction that is not transparent to typical NFV workloads: as packets progress from one NF to the other the data plane actively rewrites L2–L4 headers, which fundamentally interferes with most NFV use cases that require transparent IP forwarding. This confines us to manage NFV traffic at a higher level in the protocol stack, e.g., VXLAN, GTP, RTP. However, Istio currently does not handle these protocols natively. Accordingly, we had to resort to building our NFV prototype as an overlay with *all NF–NF traffic encapsulated in HTTP*, either natively (HTTP GET/POST) or using an HTTP-based messaging protocol (Websocket/gRPC). In this way, the original *IP 5-tuple is exposed to the service mesh in HTTP headers*, which allows to take full advantage of Istio's HTTP-based routing, security, and monitoring capabilities.

Table 3: Performance evaluation results. Dynamic: number of successful connection setup requests per second and 99th call setup latency. Static: throughput and latency measured with iperfv2 TCP/UDP test.

Chain	**Dynamic**		**Static, single-flow**			**Static, 8 flows**
	Request/sec	**Latency** (99th perc.)	**Throughput** (TCP)	**Throughput** (UDP)	**Per-Packet Latency** (avg/min/95-th/max)	**Sum Throughput**
Chain 1	235.5 qps	98 ms	100 Mbps	35 Mbps/4,375 pps	**1.37**/0.18/7.1/96.7 ms	1.24 Gbps
Chain 2	222.5 qps	88 ms	100 Mbps	25 Mbps/3,125 pps	**2.17**/0.53/13.2/97.1 ms	946 Mbps
Chain 3	184.5 qps	91 ms	100 Mbps	20 Mbps/2,500 pps	**4.58**/1.03/44.1/96.7 ms	655 Mbps

Minikube v1.0.0, Kubernetes v1.14.0, Istio v1.1.5, Virtualbox: 4 x Intel Xeon CPU E5-2620v3 @ 2.40GHz, HT enabled, 32 Gbyte DRAM, iperf v2.0.12, Fortio v1.3.2pre: load test, max qps, 6 threads.
Static Rate: TCP: Read buffer size: 128 KB, window size: 85.3 KB / UDP: 500 byte/packet, avg latency: <40 ms / Static Latency: UDP, 1.6 Mbps @ 200 byte packets, 1000 pps
Single-flow: from Host to Minikube VM via NodePort back to the Host / Concurrent: 8 TCP flows between two pods provisioned in the mesh, via the Istio ingress gateway

Ingress/egress protocol conversion. Typically, user traffic is posed to the NFV framework, and consumed from the system, encapsulated in use-case specific "niche" protocols, e.g., UDP, SCTP, RTP, that Istio currently does not understand. Correspondingly, a gateway functionality is necessary which *maps user traffic into, and out from, the service mesh*, transparently converting from conventional encapsulations to HTTP and *vice versa*. Some of this gateway functionality can be readily implemented with standard Linux CLI tools; in our prototype we used `socat(1)` [45], `websocat`, and `httptunnel(1)` for this purpose.

Conntrack, a first class citizen in L7 networking. In L7–SDN *the main attachment point of NFs is BSD sockets* and not plain ports as in "conventional" SDN [55]: each NF terminates HTTP sockets and either originates new HTTP sessions (*proxy mode*) or sends the processed traffic back in the HTTP response (*server mode*). Connection requests received at the gateway will trigger the establishment of a new HTTP session through the service mesh, building per-session conntrack state at session creation time. Handling connectionless (per-packet) transport, such as ICMP, is difficult in this design; our prototype falls back to tunneling this traffic in aggregate HTTP tunnels and letting the NFs do classification, header parsing, and basic IP processing by themselves.

Evaluation. We coded our prototype in about 600 lines of Javascript/Node.js and we implemented a simple WAN acceleration service with 3 NFs: encrypt-decrypt, compress-decompress, and passthrough. We deployed our prototype into a Minikube Kubernetes cluster alongside a full installation of Istio and we configured 3 service chains: `Chain 1` is an empty chain traversing only the ingress and egress gateways to test plain ingress and egress mapping to and from HTTP; `Chain 2` implements a full `AES-256` payload encrypt-decrypt cycle on top of the empty chain; and `Chain 3` extends the latter with a back-to-back gzip compress-decompress cycle. The details of the configurations and the results are shown in Table 3.

Depending on the workload, our L7–NFV prototype can *set up hundreds of user calls per second* with at most *100 msec latency per call setup*, providing *100 Mbps rate in TCP* (stream) and about *a third of that on UDP* (datagram) traffic. The prototype delivers 1–5 *ms average per-packet latency*, but with a fairly large uncertainty envelope of 100 ms; this is clearly attributed to the inefficiencies of the sidecar proxy model. The workload *elastically scales to* 8 *concurrent flows* reaching an aggregate throughput in the order of gigabits per second. This seems enough for request-response and unsegmented byte-stream workloads, but for per-packet datagram workloads the measured $5,000$–$10,000$ pps (packet per second) throughput is orders of magnitude short of the performance we have grown to expect from a telco-grade NFV framework [19, 25, 33, 34, 39, 51].

The Good. The application-layer approach provides a number of unique advantages in L7–NFV. First, the design and implementation of NFs is simplified a lot compared to the L2–L4 case; in L2–L4 each NF needs a separate classifier to sort input traffic into per-user flows [3], while in L7–NFV this is done automatically by the underlying network stack. Correspondingly, the full implementation of the 3 NFs takes only 15 SLOC(!) of Javascript/Node.js. Second, traffic management is smooth at L7 as there is no need to worry about dynamic IP addresses and routing tables; this is automatically handled by Kubernetes/Istio. Third, many of the resiliency, security, and elastic scaling features that in NFV currently require additional frameworks [34, 52] and extra management burden [43] are provided off-the-shelf by Istio. Finally, notorious pain points in "traditional" NFV [43], like maintaining session affinity [40] and session identity across NFs, load-aware load-balancing [27], or in-workload encryption, are particularly easy at L7 and come natively supported in Istio.

The Bad. Some of the design decisions we had to make (recall Table 2) adversely affect the present viability of L7–SDN on top of Istio. First, application-layer network virtualization would at the minimum require the data plane to implement said application-layer protocols. Unfortunately, Istio's lack of support for basically any L7 protocol beyond HTTP, and above all, everything running on top of UDP, required us to bolt the entire NFV framework on top of HTTP, which proves a poor fit for datagram workloads. Furthermore, while off-the-shelf offerings in Kubernetes/Istio are unmatched when it comes to monitoring and tracing, these seem of limited use when the KPIs we can observe are mostly meaningless in the NFV context; e.g., Istio KPIs are "requests per second" and "call setup latency", while in NFV we are concerned with throughput and per-packet latency. Similar is the case for enforcing Service Level Objectives and Quality of Service. Nonetheless, we believe these problems would be feasible to address *within* the existing service-mesh frameworks with sufficient effort.

The Ugly. Some problems, however, seem deeply immersed in the current design of Istio and the split-stack SDN architecture. Since the targeted workload of the service mesh is short-lived HTTP request-response communication, the typical sidecar proxy does not track and expose individual sessions to the control plane. This means that there is no way to list or kill individual user sessions, and one cannot dynamically reroute or re-connect a live session. Moreover, we found that there is limited support in Istio/Envoy for basic telco data plane functionality [25], like encapsulation/

decapsulation, stream multiplexing/de-multiplexing, or explicit protocol conversion, and adding these may easily amount to rewriting Envoy from scratch. But perhaps most compellingly, Envoy's data plane is closely and intricately tied to the OS socket interface and thus the usual NFV performance-boosting tricks, like fast user-space network stacks [1, 14, 41], cannot be used under Istio. This is exacerbated by the inefficiencies of the sidecar proxy model, enforced by the "split-stack" design.

Takeaway. An L4–L7 approach to NFV has a number advantages over conventional L2-L3 approaches, most notably, simplified NF development and native support for some of the pain-points in traditional NFV. Unfortunately, the restrictions inherent to the state-of-the-art L7–SDN design enforce a number of compromises that seem to greatly hinder its applicability to network-intensive use cases, like NFV. Addressing these issues would require significant innovation in data plane abstractions and new control plane models.

5 TOWARDS A CLEAN-SLATE FULL-STACK SDN

We believe that a full-stack SDN architecture would be a plausible way to overcome the current limitations of L7 SDN frameworks.

We envision the *full-stack SDN switch* to be able to terminate and originate traffic at any level of the protocol stack, e.g., a raw L2–L3 datagram socket bound to a local port for receiving and sending pure Ethernet or IP packets, an unconnected datagram socket (UDP) or connected stream socket (TCP, SCTP) at L4, or an L7 session (RTP, WebSocket, etc.). The switch would then be able to transparently pipe any two streams together, driven by a set of match-action rules programmed into it from the control plane. The match-action rules in turn should work on a per-packet (for unconnected datagram sockets) or a per-connection (for connected streams) basis. To maximize efficiency, the switch should be able to dynamically offload the pipeline partially or as a whole to hardware accelerators [2, 6, 24].

Since it is not practical to wire-in all possible application-layer protocols into the data plane, a full-stack SDN switch should be able to dynamically accept configuration for defining new protocols. This would make it possible, e.g., to implement VXLAN on top of a primitive UDP stream on-the-fly or build full MPLS support from raw Ethernet frames, the same way as P4 allows to compile new protocols into the data plane dynamically.

We envision the *full-stack SDN control plane* to work either in legacy "split mode", where different controllers are responsible for different layers of the protocol stack as in today's data center virtualization, or in a "full-stack mode" where a single controller framework exerts full control and performs "cross-layer optimization" on the entire pipeline. In both modes, the control plane should be able to authorize access to different portions/layers of the pipeline and to manage overlapping or conflicting configurations.

6 RELATED WORK

The motivations for a full-stack SDN are many and well-documented: declarative L4–L7-aware traffic management [19, 33, 47], policy enforcement [26] and monitoring [29]; seamless application scale-out [17, 52] and legacy applications support [53]; simplified application side packet classifiers [3]; reduced configuration complexity eliminating the need to "micro-manage" NF chains at L2–L4 [43] or virtualized data-center networks separately at different layers of the protocol stack [22]; and value-added L7 network services, including adaptive load-balancing [27], timeouts/retries/circuit breakers, transparent encryption [34, 37], or CI/CD support.

Currently, the vision of full-stack SDN is largely unfulfilled. In the data plane, the well-known dynamically configurable service proxies, like Envoy, Traefik, HAProxy or `nginx`, are constrained to HTTP and TCP and completely lack the abstractions for native stream processing. In the control plane, Kubernetes/Istio/xDS offer good models for an extensible L7 SDN. However, to realize the full-stack SDN at its complete feature set, radically new ideas and abstractions and substantial further research is needed.

In the IT world, the enterprise service bus (ESB) and API gateways were the first attempts at an application-aware universal communication framework [38]. Meanwhile, the research community also observed the need to extend the network data plane towards L4–L7 processing: Slim implements an accelerated socket-layer network virtualization framework [54]; the congestion control plane splits the transport layer into a fast data-plane and a separate congestion control logic in a framework that can be best described as an L4–SDN [30]; and XTRA [2], PicNIC [24] and NICA [6] address the need to efficiently offload reconfigurable L4 network functions to SmartNICs.

More recently, Trident has called for a unified L2-L7–SDN control plane and defined a set of abstractions and algorithms to implement this vision, but the data plane is still assumed to be at L2–L4 [9]. In the context of NFV, the need for full-stack L2–L7 networking was raised in [53] and [5], but these works are hard to generalize beyond the setting of NFV.

ACKNOWLEDGMENTS

This research is (in part) supported by the UK's Engineering and Physical Sciences Research Council (EPSRC) under the EARL project (EP/P025374/1), and project no. 123957, 129589 and 124171 provided by the National Research, Development and Innovation Fund of Hungary under the FK-17, KH-18 and K-17 funding schemes. The authors wish to thank Balázs Sonkoly, Felicián Németh, Péter Megyesi, and Máté Nagy for the useful input they provided to the paper.

REFERENCES

[1] Advanced Networking Lab/KAIST. Packet I/O Engine. https://github.com/PacketShader/Packet-IO-Engine.
[2] G. Bianchi, M. Welzl, A. Tulumello, F. Gringoli, G. Belocchi, M. Faltelli, and S. Pontarelli. XTRA: Towards portable transport layer functions. *IEEE Transactions on Network and Service Management*, PP:1–1, 10 2019.
[3] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: a software-defined framework for developing, deploying, and managing network functions. In *ACM SIGCOMM*, pages 511–524, 2016.
[4] M. Dalton et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *USENIX NSDI*, pages 373–387, 2018.
[5] L. Dunbar, R. Parker, N. So, and D. E. Eastlake. Architecture for Chaining Legacy Layer 4-7 Service Functions. Internet-Draft draft-dunbar-sfc-legacy-l4-l7-chain-architecture-05, Internet Engineering Task Force, July 2014. Work in Progress.
[6] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *USENIX ATC*, pages 345–362, 2019.

[7] D. Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *USENIX NSDI*, pages 51–66, 2018.

[8] T. Fromm. Performance benchmark analysis of Istio and Linkerd. https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd, 2019.

[9] K. Gao, T. Nojima, and Y. R. Yang. Trident: Toward a unified SDN programming framework with automatic updates. In *ACM SIGCOMM*, pages 386–401. ACM, 2018.

[10] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data plane performance diagnosis of TCP. In *ACM SOSR*, pages 61–74. ACM, 2017.

[11] T. Graf. Accelerating Envoy and Istio with Cilium and the Linux kernel. CNCF KubeCon, 2018.

[12] Guo et al. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *ACM SIGCOMM*, 2015.

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX NSDI*, volume 11, pages 22–22, 2011.

[14] Intel. Data Plane Development Kit. http://dpdk.org.

[15] Istio: Connect, secure, control, and observe services. https://istion.io.

[16] Kubernetes: Production-grade container orchestration. https://kubernetes.io.

[17] M. Kablan, A. Alsudais, E. Keller, and F. Le. Stateless network functions: Breaking the tight coupling of state and processing. In *USENIX NSDI*, pages 97–112, 2017.

[18] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *USENIX NSDI*, 2019.

[19] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX NSDI*, pages 171–186, 2018.

[20] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[21] M. Klein. The universal data plane API. https://blog.envoyproxy.io/the-universal-data-plane-api-d15cec7a, 2017.

[22] T. Koponen and others. Network virtualization in multi-tenant datacenters. In *USENIX NSDI*, pages 203–216, 2014.

[23] S. G. Kulkarni, G. Liu, K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu. REINFORCE: achieving efficient failure resiliency for network function virtualization based services. In *ACM CoNEXT*, pages 41–53, 2018.

[24] P. Kumar, N. Dukkipati, N. Lewis, Y. Cui, Y. Wang, C. Li, V. Valancius, J. Adriaens, S. Gribble, N. Foster, and A. Vahdat. PicNIC: Predictable virtualized NIC. In *ACM SIGCOMM*, pages 351–366, 2019.

[25] T. Lévai, G. Pongrácz, P. Megyesi, P. Vörös, S. Laki, F. Németh, and G. Rétvári. The price for programmability in the software data plane: The vendor perspective. *IEEE Journal on Selected Areas in Communications*, 36(12):2621–2630, Dec. 2018.

[26] L. MacVittie. What does "HTTP is the new TCP" mean for you. F5 dev/central, 2014. https://devcentral.f5.com/s/articles/what-does-http-is-the-new-tcp-mean-for-you.

[27] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful Layer-4 load balancing fast and cheap using switching ASICs. In *ACM SIGCOMM*, pages 15–28, 2017.

[28] W. Morgan. What's a service mesh? and why do i need one? https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one, 2017.

[29] J. Nam, J. Seo, and S. Shin. Probius: Automated approach for VNF and service chain analysis in software-defined NFV. In *ACM SOSR*, pages 1–13, 2018.

[30] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan. Restructuring endpoint congestion control. In *ACM SIGCOMM*, pages 30–43. ACM, 2018.

[31] Network Service Mesh: An L2/L3 service mesh for Kubernetes. https://networkservicemesh.io.

[32] Open Hub. Discover, Track and Compare Open Source.

[33] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A framework for NFV applications. In *ACM SOSP*, pages 121–136, 2015.

[34] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the v out of NFV. In *USENIX OSDI*, pages 203–216, 2016.

[35] L. Peterson. Http is the new narrow waist. Systems Approach Blog, 2019. https://www.systemsapproach.org/blog/http-is-the-new-narrow-waist.

[36] B. Pfaff et al. The design and implementation of Open vSwitch. In *USENIX NSDI*, pages 117–130, 2015.

[37] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. Safebricks: Shielding network functions in the cloud. In *USENIX NSDI*, pages 201–216, 2018.

[38] C. Posta. Application network functions with ESBs, API management, and now, service mesh? https://blog.christianposta.com/microservices/application-network-functions-with-esbs-api-management-and-now-service-mesh, 2017.

[39] Z. A. Qazi, M. Walls, A. Panda, V. Sekar, S. Ratnasamy, and S. Shenker. A high performance packet core for next generation cellular networks. In *ACM SIGCOMM*, pages 348–361, 2017.

[40] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *USENIX NSDI*, pages 227–240, 2013.

[41] L. Rizzo. Netmap: A novel framework for fast packet i/o. In *USENIX ATC*, pages 9–9, 2012.

[42] D. Schenck. Istio dark launch: Secret services. https://developers.redhat.com/blog/2018/04/17/istio-dark-launch-secret-services, 2018.

[43] S. Shenker, S. Ratnasamy, and C. Polychronopoulos. Accelerating innovation with Lean NFV. ONS Keynote, 2019.

[44] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *ACM SIGCOMM*, pages 13–24, 2012.

[45] socat: Multipurpose relay. http://www.dest-unreach.org/socat.

[46] J. Song. Understanding how envoy sidecar intercept and route traffic in istio service mesh. Medium, 2019.

[47] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu. NFP: Enabling network function parallelism in NFV. In *ACM SIGCOMM*, pages 43–56, 2017.

[48] The OpenStack project. OpenStack Neutron integration with OVN. https://docs.openstack.org/networking-ovn/latest.

[49] K. Thimmaraju, S. Hermak, G. Rétvári, and S. Schmid. MTS: Bringing multi-tenancy to virtual networking. In *USENIX ATC*, pages 521–536, 2019.

[50] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid. Taking control of SDN-based cloud systems via the data plane. In *ACM SOSR*, pages 1–15, 2018.

[51] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker. Resq: Enabling SLOs in network function virtualization. In *USENIX NSDI*, pages 283–297, 2018.

[52] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker. Elastic scaling of stateful network functions. In *USENIX NSDI*, pages 299–312, 2018.

[53] C. Zhang, S. Addepalli, N. Murthy, L. Fourie, M. Zarny, and L. Dunbar. L4-L7 service function chaining solution architecture. ONF TS-027, 2015.

[54] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. Slim: OS kernel support for a low-overhead container overlay network. In *USENIX NSDI*, pages 331–344, 2019.

[55] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. Slim: OS support for a low-overhead container overlay network. In *USENIX NSDI*, pages 331–344, 2019.