Retrospective on "Towards an Active Network Architecture"

David Wetherall
Google
wetherallx@google.com

David Tennenhouse
VMWare
tennenhouse@vmware.com

This article is an editorial note submitted to CCR. It has NOT been peer reviewed.
The authors take full responsibility for this article's technical content. Comments can be posted through CCR Online.

ABSTRACT

Network programmability has metamorphosed over the past twenty years from the controversial research vision of active networks, through PlanetLab, to the juggernaut of SDN and OpenFlow that has swept industry. Now PISA switches are emerging with support for protocol-independent reconfigurability. We reflect on how network architecture has evolved along a different path than we had foreseen to arrive at a place that is not so different than we and other researchers had hoped and imagined.

KEYWORDS

Active Networks, OpenFlow, SDN.

1 INTRODUCTION

> *...packets contain header fields interpreted as "telemetry instructions" by network devices. These instructions tell an INT-capable device what state to collect and write into the packet as it traverses the network. [17]*

The quote above would be at home in an active network paper from the late 1990s describing a research vision. It is drawn instead from the P4.org In-band Network Telemetry (INT) specification in 2018, at a time when switch vendors are beginning to support data plane telemetry. The journey over the intervening twenty odd years is already described well in articles such as "The Road to SDN: An Intellectual History of Programmable Networks" by Feamster, Rexford and Zegura [11]. We provide a brief summary of this transformation next for the sake of a standalone paper. But our purpose in this article is to reflect with some humility on how network architecture has evolved in a different way than we had foreseen to arrive at a place where active networks remain highly relevant to the network architecture. We also remind the reader of other retrospective articles on active networks, such as Calvert's "Reflections on Network Architecture" [4].

The thinking on network programmability that began as active networks has crystallized in network devices that provide flexibility independent of the particulars of data plane protocols by using standards such as OpenFlow [13], P4 [7], and OpenConfig [16]. A capsule summary of network programmability is as follows[1]. Active networks [20] was motivated by the ongoing need to extend the functionality that is logically performed within the network, plus the tortuously slow path of doing so via IETF protocol standards. The lengthy delays and high barriers to extending network functionality motivated further systems such as PlanetLab [6], and have continued to be a problem to the present day. The bold vision of active networks was to use some form of programmability as the answer to both problems. Two forms were explored: one in which packets literally invoked programs hosted by the infrastructure [21]; and the other in which operators injected programs to control their infrastructure [1].

Active networks are mostly remembered for the first, more extreme vision, though it is the latter form, championed by Jonathan Smith, that has proved far more practical, at least to date. The response by the network community was much controversy over embedding potentially complex functions in the network architecture [5], and an intellectual flowering of research. Yet there was no direct transfer of results to industry, as the lack of a driving use case stalled progress and adoption.

Subsequent efforts at programmability focused on the separation of control and data planes. Traditional routers are vertically integrated, handling tasks from basic forwarding to full-blown routing protocols. Separation provides twin advantages for operators. Logically centralizing the control plane functions facilitates network-wide control over management policies, notably traffic engineering. And reducing switches to forwarding devices simplifies their operation. OpenFlow emerged by the late 2000s as an enabling API [13] for this form of programmability. It led to what is now widely called SDN (Software-Defined Networking), in which switches are simple forwarding engines that are controlled by more complex management software.

The next chapter in network programmability is embodied in the evolution of SDN from OpenFlow to P4 [3] in the mid 2010s. OpenFlow controls switch behavior based on the pragmatic model of a classical forwarding table. While more general than IP switches of the past, it is a familiar model. On the other hand, P4 allows data plane forwarding to be specified in the form of a high-level program, which is one of the hallmarks of active networks. The language is "protocol independent" meaning that new protocols can be expressed within the framework of P4 regardless of their relationship to IP. INT is an example of a new protocol.

We have come full circle in our story: switches are becoming flexible forwarding devices whose functionality can be redefined by the operator using a high-level programming language. This is exactly the kind of outcome active networks sought to achieve.

2 REFLECTIONS

We've been privileged to witness the realization of network programmability in the form of SDN over the past decade. However, other trends such as datacenters and VMs have been the engines of change for the network landscape; programmability is one tool for effecting it.

[1] Pun intended. We hope some of you will at least chuckle.

Datacenter Networking Economics

Active networks targeted the problem of readily introducing new network functions with the research community in mind [20]. OpenFlow was similarly conceived with campus network experimentation in mind [15]. But the key driver for the adoption of SDN by industry was (of course) economics, not programmability.

Large and growing costs came with the tremendous scaling of datacenters, starting in the 2000s and continuing to this day. The major players like Google, Microsoft, Amazon, and Facebook have invested many billions of dollars to build ever-larger networks, which in turn has emphasized ways to save on cost. Traditional networking equipment used by network operators for their backbone networks did not fit the datacenter role well. These devices are expensive, complex, and limited in scale.

Separating the control and data planes allowed for improvements in these factors, a story told in Google's datacenter networking retrospective [19]. The control plane can be stripped down and customized by the operator. It's software. The data plane can be implemented with many simpler, less expensive devices. That's commodity hardware. It has been enabled by rapid advances in merchant switching silicon in the 2000s: suddenly it was possible to buy a single ASIC that supported dozens of ports capable of Pb/s switching [10]. Standards such as OpenFlow, and more recently SAI (Switch Abstraction Interface) originating from Microsoft, fit into this picture by letting third-party software control switch forwarding in a vendor-neutral manner.

This structure works well because it aligns incentives and the ability to execute, including independent adoption by datacenter operators. We did not see this coming, but recognize it to be an excellent vehicle for innovation. Success in the datacenter opened the door to the WAN, with large providers such as Google and Azure now having software-defined backbones. We also see wireless systems such as 5G moving in the same direction.

Network Virtualization

Virtual Machines (VMs) have changed computing. They are also changing networking because of the natural synergy with network virtualization, i.e., running multiple distinct networks on a common substrate. Thus they have been an important use case for network programmability. A key example is multi-tenant datacenters, in which each tenant runs a set of VMs. Each tenant must see their own network, independent of the other tenants. A natural way to accomplish this isolation is to control the path of tenant traffic through devices. Since these paths extend to VMs, the programming of flows extends to hosts. Systems such as Open vSwitch (OVS) [18] are designed expressly for this purpose.

Encapsulation strategies have long been used in networking to stitch together logically separate networks that coexist with each other. They were an original target of active networks, which relied heavily on overlays. PlanetLab improved virtualization support with slices [6]. Now, driven by VMs, there is a large and growing set of protocols that support network virtualization behaviors as the community works out the best solutions. The protocols range from classic GRE and VLANs to newer standards for VXLAN, NVGRE and GENEVE. However, more than a protocol standard, what we need are methods to manage entire networks in the manner of SDN. What better use of network programmability in both the control and data planes than to support this exploration and transition, with programming languages providing a systematic and rigorous approach?

The Pragmatism of OpenFlow

Without ASIC support, network programmability is confined to hosts running at the edge of the network; there have been multiple attempts at using servers as switches for flexibility [8], but the resulting performance pales in comparison to silicon. Thus one of the largest steps forward in network programmability was surely OpenFlow, combined with the singular force that is Nick McKeown.

Observe that OpenFlow is a deliberately pragmatic choice for switch support of programmability [13]. The API essentially provides a way for external programs to manipulate entries in the switch forwarding table. This functionality was at the core of what switch chips already did using routing protocols and other controls. Wrapping this functionality with an API provides a viable path for device support from switch vendors (especially in the world of commodity switches and merchant silicon). There was no code that was injected to switches and no need to change switch silicon. Yet simply providing this API enabled new software systems to manipulate devices to stitch together entire networks for purposes like datacenters and virtualization. This development was very timely because it came when operators were grappling with altering routing protocols for purposes such as traffic engineering. OpenFlow provided the ability to control pathing directly rather than, for example, by adjusting link weights.

The evolution to richer forms of programmability is a more recent development. OpenFlow is tied to the formats of commonly-used protocols, like TCP. The ability to reconfigure the hardware to support new protocols with different formats required a generalization of the traditional switch architecture into a PISA (Protocol Independent Switch Architecture) switch. This is a relatively recent development where the value of programmability will be proved out (or not). Reconfigurable Match Tables (RMT) [2] are one leading candidate for this generalization for switch chips, along with the P4 language [3] for programming a PISA switch. The initial mover in this space is Intel/Barefoot with their Tofino series of merchant silicon switching chips that implement P4.

Speeding Evolution

Programmability is valuable to speed what is otherwise an unworkably long process of network evolution, but it is not the goal *per se*. The driving need is to support future network use cases, of which there are many possibilities: gathering switch telemetry; redirecting and load balancing traffic; measuring link usage and congestion for host transports; support for multicast and content distribution; sender/receiver rendezvous; detection of incast patterns; and network filtering. Several of these domains require information from within the network to function properly. This is especially the case for security functions like DDOS protection. Here, unwanted traffic aggregates must be detected and filtered in the network by switches, because the damage is done if the traffic reaches hosts.

So the key question is: what kind of switch programmability is effective for supporting future use cases? This question has been

controversial since the days of active networks, though early debates on the relationship with the end-to-end argument [5] now seem quaint. Several points seem clear to us with the benefit of hindsight:

- We should prefer to run network functions as close to the edge of the network as feasible to simplify the core of the network.
- Some functions nonetheless need to run within the network, including the core of the network, because they depend on network state
- Flexible support for functions that can access protocol fields and run in the network is an enabler of network evolution, even when the set is small.

The first point fits well with the separation of control and data planes in SDN: run the more complex control plane on hosts and a lean data plane on switches. The reason is that modern network switches have essentially no buffering relative to their speeds. Thus any operation that requires non-trivial memory or delay or otherwise adds complexity will be preferentially pushed from the switch data plane and towards hosts. This includes the implementation of network-wide policies and all manner of proxies.

The second point notes that there are often corresponding functions that are needed at switches to support host policies and proxies. They are primitives that cannot be provided at hosts because they require switch state. Examples include QOS, pathing decisions for load balancing, and telemetry. Low-complexity algorithms exist for all of these examples.

The final point says that the switch support is crucial even though it is small in functionality compared with what runs on hosts. And to be able to evolve it, limited forms of programmability are highly effective because they admit many combinations of primitives across all manner of packet formats. INT [17] provides an example of how programmability can add value. With a small set of "telemetry instructions", INT lets hosts send traffic that optionally extracts telemetry from switches that are programmed with P4 (or otherwise) to be INT-capable. This telemetry can only come from switches. It already existed there and programmability makes it accessible. Hosts can extract and use this telemetry in many ways that are tailored to their needs. For example, a host may want to trace the path of a TCP connection that is experiencing retransmissions to pinpoint the location of queuing and loss. The host can simply encode INT packets to do this, collecting switch telemetry in packets that continue to carry payloads.

While this example may seem minor, it captures the kind of capability that we fundamentally need and lack today. Because we cannot get the switch telemetry we need with established protocols, we make do with workaround tools such as traceroute and pathchar [9]. They are very clever workarounds, to be sure, but they are approximations rather than a reliable way to solve the problem. We should do better, and with programmability we can.

The Promise of Programming

Active networks emphasized the use of programming languages because of the benefits they bring via program analysis as well as expressiveness. Type-checking and static analysis help with testing and the verification of high-level invariants, e.g., all packets have a TTL or equivalent field whose use prevents long-lived packets. Not only individual switches, but entire networks can be verified for properties. Ultimately, declarative specifications of network behavior can be compiled into programs for individual switches.

The use of programming language techniques promises to put networks on a computer science foundation that complements the usual engineering considerations. For example, in large networks some components are failed and being repaired, some are drained for maintenance, some have been upgraded while others have not, and some are being turned up or down. The network is always in transition. Since the combinatorial state space is enormous, it is easy for traditional designs to experience occasional, unwanted side-effects in which a less preferred path is taken, a hotspot is created, or, worse, connectivity is broken. These problems are often time-consuming to debug. Instead, it is far better if we are able to prevent them by design using program analysis.

This promise is largely unmet today and remains of substantial interest to the research community. In the host setting, domain-specific languages such as eBPF [14] have already proven effective, and are enjoying broader use as a mechanism to extend network functionality. We expect to see renewed progress on this front as reconfigurable, protocol-independent switches reach the market. As SDN was the driver for OpenFlow and programmability was the tool, we will need a driver for which program analysis is the tool if we are to realize significant progress in practice. We do not know what that driver will be, but do know that the complexity of operating networks presents a rich target as it remains too high.

3 CONCLUSION

At the time of active networks, the Internet was regarded as ossified, a victim of its own success that made change at the network level impossibly slow. Yet within a decade, trends such as SDN and network virtualization have wrought a sea change in unlocking innovation. An important legacy of active networks is the body of students and papers that have come out of it as an intellectual inquiry. We would like to think that these students and papers helped to prepare the ground for the advances in network programmability and the innovation in network architecture that followed.

And there is much more to come. We are at an exciting time as reconfigurable, protocol-independent switches begin to enter the market. Host and NIC functionality is becoming part of the network that may be manipulated. And at switches primitives such as Segment Routing [12] are generalizing the forwarding capability. We do not know what the future holds, but anticipate that constrained forms of programmability will play a large and growing role.

ACKNOWLEDGMENTS

This note represents the opinions of the authors and does not represent the views of their respective employers. We thank Jeff Mogul for feedback.

REFERENCES

[1] D. S. Alexander et al. 1998. The SwitchWare Active Network Architecture. *Netwrk. Mag. of Global Internetwkg.* 12, 3 (May 1998), 29–36.
[2] Pat Bosshart et al. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. *SIGCOMM* 43, 4 (Aug. 2013), 99–110.
[3] Pat Bosshart et al. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM* 44, 3 (July 2014), 87–95.

[4] Ken Calvert. [n. d.]. Reflections on Network Architecture: An Active Networking Perspective. *SIGCOMM Comput. Commun. Rev.* (April [n. d.]).

[5] T. M. Chen and A. W. Jackson. 1998. Commentaries on "Active networking and end-to-end arguments". *IEEE Network* 12, 3 (May 1998), 66–71.

[6] Brent Chun et al. 2003. PlanetLab: An Overlay Testbed for Broad-coverage Services. *SIGCOMM Comput. Commun. Rev.* (July 2003).

[7] The P4 Language Consortium. 2018. P4 Language Specification v1.1.0. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.pdf. (November 2018).

[8] Mihai Dobrescu et al. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *SOSP*. ACM, New York, NY, USA, 15–28.

[9] Allen B. Downey. 1999. Using Pathchar to Estimate Internet Link Characteristics. In *SIGCOMM*. 241–250.

[10] Nathan Farrington, Erik Rubow, and Amin Vahdat. 2009. Data Center Switch Architecture in the Age of Merchant Silicon. In *IEEE Symposium on High Performance Interconnects (HOTI '09)*. IEEE Computer Society, Washington, DC, USA, 93–102.

[11] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Comput. Commun. Rev.* 44, 2 (April 2014), 87–98.

[12] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois. 2015. The Segment Routing Architecture. In *GLOBECOM*. 1–6.

[13] Open Networking Foundation. 2015. OpenFlow Switch Specification Ver 1.5.1. TS-025. (March 2015).

[14] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX (USENIX'93)*. USENIX Association, Berkeley, CA, USA, 2–2.

[15] Nick McKeown et al. 2008. OpenFlow: enabling innovation in campus networks. *Computer Communication Review* 38, 2 (2008), 69–74.

[16] OpenConfig. 2016. OpenConfig. (2016). http://openconfig.net

[17] P4.org. 2016. In-band Network Telemetry (INT). P4 Dataplane Telemetry Specification. (June 2016).

[18] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. 2009. Extending Networking into the Virtualization Layer. In *HotNets-VIII*.

[19] Arjun Singh et al. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*. ACM, New York, NY, USA, 183–197.

[20] David L. Tennenhouse and David J. Wetherall. 1996. Towards an Active Network Architecture. *Computer Communication Review* 26, 2 (April 1996), 5–17.

[21] David Wetherall. 1999. Active Network Vision and Reality: Lessons from a Capsule-based System. In *SOSP (SOSP '99)*. ACM, New York, NY, USA, 64–79.