Fabric: A Retrospective on Evolving SDN

Martín Casado* Teemu Koponen* Scott Shenker†‡ Amin Tootoonchian§†

Abstract

MPLS was an attempt to simplify network hardware while improving the flexibility of network control. Software-Defined Networking (SDN) was designed to make further progress along both of these dimensions. While a significant step forward in some respects, it was a step backwards in others. In this paper we discuss SDN's shortcomings and propose how they can be overcome by adopting the insight underlying MPLS. We believe this hybrid approach will enable an era of simple hardware and flexible control.

1 Introduction

The advent of the Internet, and networking more generally, has been a transformative event, changing our lives along many dimensions: socially, societally, economically, and technologically. While the overall architecture is an undeniabe success, the state of the networking industry and the nature of networking infrastructure is a less inspiring story. It is widely agreed that current networks are too expensive, too complicated to manage, too prone to vendor-lockin, and too hard to change. Moreover, this unfortunate state-of-affairs has remained true for well over a decade. Thus, while much of the research community has been focusing on "clean-slate" designs of the overall Internet *architecture*, a more pressing set of problems remain in the design of the underlying network *infrastructure*. That is, in addition to worrying about the global Internet protocols, the research community should also devote effort to how one could improve the infrastructure over which these protocols are deployed.

This infrastructure has two components: (i) the underlying hardware and (ii) the software that controls the overall behavior of the network. An ideal network design would involve hardware that is:

- **Simple:** The hardware should be inexpensive to build and operate.
- **Vendor-neutral:** Users should be able to easily switch between hardware vendors without forklift upgrades.
- **Future-proof:** The hardware should, as much as possible, accommodate future innovation, so users need not upgrade their hardware unnecessarily.

The ideal software "control plane" coordinating the forwarding behavior of the underlying hardware must meet a single but broad criterion:

- **Flexible:** The software control plane should be structured so that it can support the wide variety of current requirements (such isolation, virtualization, traffic engineering, access control, etc.) and, to the extent possible, be capable of meeting to unknown future requirements as they arise.

Today's networking infrastructure does not satisfy *any* of these goals, which is the cause of significant pain for network operators. In fact, in terms of impact on user experience, the inadequacies in these infrastructural aspects are probably more problematic than the Internet's architectural deficiencies.

The inability to meet these goals is not for lack of trying: the community has repeatedly tried new approaches to network infrastructure. Some of these attempts, such as Active Networking [22], focused more on flexibility than practicality, while others, such as ATM [7], had the opposite emphasis; out of a long list of ephemeral and/or ineffective proposals, by far the most successful approach has been MPLS [18]. MPLS is now widely deployed and plays a crucial role in VPN deployment and traffic engineering. While originally decried by some as an architectural abomination, we will argue later that MPLS embodies an important insight that we must incorporate in future network designs.

However, MPLS did not meet all the goals of an ideal network, so more recently the community has made another attempt at reaching networking nirvana: Software-Defined Networking (SDN) [10, 12, 13, 16]. There has been a staggering level of hype about SDN, and some knee-jerk opposition; the shallowness of the discussion (which occurs mostly in trade magazines and blogs) has largely overlooked SDN's more fundamental limitations.

In this paper we discuss these limitations and explain how SDN, by itself, would fall short of the goals listed above. We

*Nicira
†International Computer Science Institute (ICSI)
‡UC Berkeley
§University of Toronto

then describe how we might create a better form of SDN by retrospectively leveraging the insights underlying MPLS. This modified approach to SDN revolves around the idea of network *fabrics*[1] which introduces a new modularity in networking that we feel is necessary if we hope to achieve both a simple, vendor-neutral, and future-proof hardware base and a sufficiently flexible control plane.

We hasten to note that this paper contains no new technical research, as fabrics are already well established in the academic and commercial arenas (see, for example, [2, 11, 15]). However, while it would be easy to dismiss what we write here as "nothing new", note that the direction we propose for SDN is quite different from what is being pursued by current ONF standardization efforts and what is being discussed in the academic SDN literature. Thus, our paper should be read not as a deep technical contribution but as a "call to arms" for the SDN community to look backwards towards MPLS as they plan for SDN's future.

We begin this paper (Section 2) by reviewing the basics of traditional network design, MPLS, and SDN. We then, in Section 3, introduce the a hybrid approach that combines SDN and MPLS. We end in Section 4 with a discussion of the implications of this approach.

2 Background on Network Designs

2.1 Overview

In designing the network infrastructure, it is important to consider network *requirements* and network *interfaces*. Network requirements come from two sources: hosts and operators. Hosts (or, more accurately, the users of that host) want their packets to travel to a particular destination, and they may also have QoS requirements about the nature of the service these packets receive en route to that destination. Network operators have a broader set of requirements — such as traffic engineering, virtualization, tunneling and isolation — some of which are invisible and/or irrelevant to the hosts. As we observe below, the control mechanisms used to meet these two sources of requirements are quite different (for largely historical reasons).

Like any system, networks can be thought of in terms of interfaces; here we use that term not to refer to a formal programmatic interface, but to mean more generally and informally places where control information must be passed between network entities. There are three relevant interfaces we consider here:

- *Host — Network:* The first interface is how the hosts inform the network of their requirements; this is typically done in the packet header (for convenience, in the following we will focus on L3, but our comments apply more generally), which contains a destination address and (at least theoretically) some ToS bits. However, in some designs (such as IntServ), there is a more explicit interface for specifying service requirements.
- *Operator — Network:* The second interface is how operators inform the network of their requirements; traditionally, this has been through per-box (and often manual) configuration, but SDN (as we discuss later) has introduced a programmatic interface.
- *Packet — Switch:* The third interface is how a packet identifies itself to a switch. To forward a packet, a router uses some fields from the packet header as an index to its forwarding table; the third interface is the nature of this index.

We now turn to how the original Internet, MPLS, and SDN deal with requirements and implement these interfaces.

2.2 Original Internet

In the original Internet design, there were no operator requirements; the goal of the network was to merely carry the packet from source to destination (for convenience, we will ignore the ToS bits), and routing algorithms computed the routing tables necessary to achieve that goal. At each hop, the router would use the destination address as the key for a lookup in the routing table; that is, in our conceptual terms, every router would independently interpret the host requirements and take the appropriate forwarding action. Thus, the Host-Network and Packet-Switch interfaces were identical, and there was no need for the Operator-Network interface.

2.3 MPLS

MPLS introduced an explicit distinction between the network edge and the network core. Edge routers inspect the incoming packet headers (which express the host's requirements as to where to deliver the packet) and then attach a label onto the packet which is used for all forwarding within the core. The label-based forwarding tables in core routers are built not just to deliver packets to the destination, but to also address operator requirements such as VPNs (tunnels) or traffic engineering. MPLS labels have meaning only within the core, and are completely decoupled from nature of the host protocol (*e.g.*, IPv4 or IPv6) used by the host to express its requirement to the network. Thus, the interface for specifying host requirements is still IP, while the interface for packets to identify themselves is an MPLS label. However, MPLS did not formalize the interface by which operators specified their control requirements. Thus, MPLS distinguished between the Host-Network and Packet-Switch interfaces, but did not develop a general Operator-Network interface.

2.4 SDN

In contrast to MPLS, SDN focuses on the control plane. In particular, SDN provides a fully programmatic Operator-Network interface, which allows it to address a wide variety

[1] We use this term in a general sense of a contiguous and coherently controlled portion of the network infrastructure, and do not limit its meaning to current commercial fabric offerings.

of operator requirements without changing any of the lower-level aspects of the network. SDN achieves this flexibility by decoupling the control plane from the topology of the data plane, so that the distribution model of the control plane need not mimic the distribution of the data plane. While SDN in the most general terms can apply to any network design with decoupled control and data planes [3, 5, 6, 9, 20], we will frame this discussion around its canonical instantiation: OpenFlow.[2]

In OpenFlow each switch within the network exports an interface that allows a remote controller to manage its forwarding state. This managed state is a set of forwarding tables that provide a mapping between packet header fields and actions to execute on matching packets. The set of fields that can be matched on is roughly equivalent to what forwarding ASICs can match on today, namely standard Ethernet, IP, and transport protocol fields; actions include common packet operations of sending the packet to a port as well as modifying the protocol fields. While OpenFlow is a significant step towards making the control plane more flexible, it suffers from a fundamental problem that it does not distinguish between the Host-Network interface and the Packet-Switch interface. Much like the original Internet design, each switch must consider the host's original packet header when making forwarding decisions. Granted, the flexibility of the SDN control plane allows the flow entries, when taken collectively, to implement sophisticated network services (such as isolation); however, each switch must still interpret the host header.[3]

This leads to three problems:

- First, it does not fulfill the promise of simplified hardware. In fairness, OpenFlow was intended to strike a balance between practicality (support matching on standard headers) and generality (match on all headers). However, this requires a switch hardware to support a lookup over hundreds of bits; in contrast, core forwarding with MPLS needs only matches over some tens of bits. Thus, with respect to the forwarding hardware alone, an OpenFlow switch is clearly far from the simplest design achievable.

- Second, it does not provide sufficient flexibility. We expect host requirements to continue to evolve, leading to increasing generality in the Host-Network interface, which in turn means increasing the generality in the matching allowed and the actions supported. In the current OpenFlow design paradigm, this additional generality must be present on every switch. It is inevitable that, in OpenFlow's attempt to find a sweet spot in the practicality vs generality tradeoff, needing functionality to be present on every switch will bias the decision towards a more limited feature set, reducing OpenFlow's generality.

- Third, it unnecessarily couples the host requirements to the network core behavior. This point is similar to but more general than the point above. If there is a change in the external network protocols (*e.g.*, switching from IPv4 to IPv6) which necessitates a change in the matching behavior (because the matching must be done over different fields), this requires a change in the basic packet matching, even in the core of the network.

Thus, our goal is to extend the SDN model in a way that avoids these limitations yet still retains SDN's great control plane flexibility. To this end, it must retain its programmatic control plane interface (so that it provides a general Operator-Network interface), while cleanly distinguishing between the Host-Network and Packet-Switch interfaces (as is done in MPLS). We now describe such a design.

3 Extending SDN

3.1 Overview

In this section we explore how the SDN architectural framework might be extended so that it is able to provide benefits discussed previously in the paper. Our proposal is centered on the introduction of a new conceptual component which we call the "network fabric". While a common term, for our purposes we limit the definition to refer to a collection of forwarding elements whose primary purpose is packet transport. Under this definition, a network fabric does *not* provide more complex network services such as filtering or isolation.

The network then has three kinds of components (see Figure 1): hosts, which act as sources and destinations of packets; edge switches, which serve as both ingress and egress elements; and the core fabric. The fabric and the edge are controlled by (logically) separate controllers, and the edge is responsible for complex network services while the fabric only provide basic packet transport. The edge controller handles the Operator-Network interface, the ingress edge switch, along with its controller, handle the Host-Network interface, and the switches in the fabric are where the Packet-Switch interface is exercised.

The idea of designing a network around a fabric is well understood within the community. In particular, there are many examples of limiting the intelligence to the network edge and keeping the core simple.[4] Thus, our goal is not

[2]We note that there has been a wealth of recent work on SDN, including [1, 4, 8, 14, 17, 19, 21, 23–25], which extends SDN in one or more directions, but all of them are essentially orthogonal to the issues we are discussing here.

[3]One could, of course, use SDN to implement MPLS. However, each switch must be prepared to deal with full host headers.

[4]This is commonly done for example in datacenters where connectivity is provided by a CLOS topology running an IGP and ECMP. It is also reflected in WAN where interdomain policies are implemented at the provider edge feeding packets into a simpler MPLS core providing connectivity across the operator network.

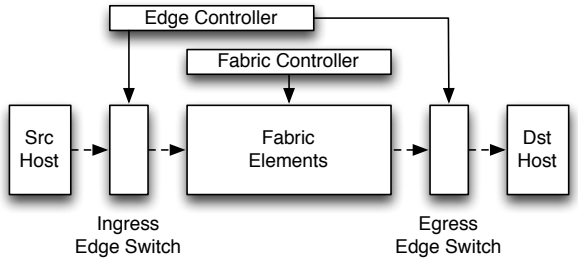

Figure 1: Source host sends a packet to an edge switch, which after providing network services, sends it across the fabric for the egress switch to deliver it to the destination host. Neither host sees any internals of the fabric. The control planes of the edge and fabric are similarly decoupled.

to claim that a network fabric is a new concept but rather we believe it should be included as an architectural building block within SDN. We now identify the key properties for these fabrics.

Separation of Forwarding. In order for a fabric to remain decoupled from the edge it should provide an interface of minimal set of forwarding primitives without exposing any internal forwarding mechanisms that would be visible from the end system if the fabric was replaced. We describe this in more detail below but we believe it is particularly important that external addresses are not used in forwarding decisions within the fabric both to simplify the fabric forwarding elements, but also to allow for independent evolution of fabric and edge.

Separation of Control. While there are multiple reasons to keep the fabric and the edge's control planes separate, the one we would like to focus on is that they are solving two different problems. The fabric is responsible for packet transport across the network, while the edge is responsible for providing more semantically rich services such as network security, isolation, and mobility. Separating the control planes allows them each to evolve separately, focusing on the specifics of the problem. Indeed, a good fabric should be able to support any number of intelligent edges (even concurrently) and vice versa.

Clearly, SDN and fabric offer many similar benefits, although in different contexts. If the interfaces to the fabric are clearly defined and standardized, it offers vendor independence. As we describe in more detail later, limiting the function of the fabric to forwarding is amenable to a much simpler switch implementations. And while these benefits are similar to those offered by SDN, they are complimentary rather than conflicting.

3.2 Fabric Service Model

Under our proposed model, a fabric is a system component which roughly represents raw forwarding capacity. In theory, a fabric should be able to support any number of edge designs including different addressing schemes and policy models. The reverse should also be true, that is, a given edge design should be able to take advantage of any fabric regardless of how it was implemented internally.

Primitive	Description
Attach(P)	Attach a fabric port P to the fabric.
Send(P, pkt)	Send a packet to a single fabric port.
Send(G, pkt)	Send a packet to a multicast group G.
Join(G, P)	Attach a port to a multicast group G.
Leave(G, P)	De-attach a port from a multicast group G.

Table 1: Fabric service model. Note that the ToS bits in the packet for QoS within the fabric are not included above.

We have found that the design of a modern router/switch chassis is a reasonably good analogy for an SDN architecture that includes a fabric. In a chassis, the line cards contain most of the intelligence and they are interconnected by a relatively dumb, but very high bandwidth, backplane. Likewise, in an SDN architecture with a fabric, the edge will implement the network policy and manage end-host addressing, while the fabric will effectively interconnect the edge as fast and cheaply as possible.

The chassis backplane therefore provides a reasonable starting point for a fabric service model. Generally, a backplane supports point-to-point communication, point-to-multi-point communication, and priorities to make intelligent drop decisions under contention. In our experience, this minimal set is sufficient for most common deployment scenarios. More complex network functions, such as filtering, isolation, stateful flow tracking, or port spanning can be implemented at the edge. Table 1 summarizes this high-level service model offered by the fabric.

3.3 Fabric Path Setup

Another consideration is path setup. In the "wild" two methods are commonly used today. In the datacenter, a common approach is to us a standard IGP (like OSPF) and ECMP to build a fabric. In this case, all paths are calculated and stored in the fabric. MPLS, on the other hand, requires the explicit provisioning of an LSP by the provider. The primary difference between the two is that when all forwarding state is precalculated, it is normally done with the assumption that any point at the edge of the fabric can talk to any other point. On the other hand, provisioned paths provide an isolated forwarding context between end points that is dictated by network operator (generally from the provider edge).

We believe that either model works in practice depending on the deployment environment. If both the edge and the fabric are part of the same administrative domain, then precalculating all routes saves operational overhead. However, if the edge and fabric have a customer provider relationship, then an explicit provisioning step may be warranted.

3.4 Addressing and Forwarding in the Fabric

As we have described it, a forwarding element in the fabric differs from traditional network forwarding elements in two ways. First, they aren't required to use end-host addresses for forwarding, and second, they are only responsible for delivering a packet to its destination(s), and not enforcing any policy. As a result, the implementation of the fabric forwarding element can be optimized around relatively narrow requirements. Two current approaches exemplify the options available:

- One option would be to follow MPLS and limit the network address to opaque labels and the forwarding actions to forward, push, pop and swap. This would provide a very general fabric that could be used by multiple control planes to provide either path-based provisioning, or even destination based forwarding with label-aggregation.
- Another option would be to limit the packet operations to a destination address lookup with a longest prefix match with ECMP-based forwarding. It is unlikely that this would be suitable for path-based provisioning, but it would likely result is a simpler control plane and higher port densities.

Our preference is to use labels similar MPLS because it supports a more general forwarding model. However, the main point of this paper is that the SDN architecture should support incorporate the notion of a fabric, but SDN need not be concerned with the specifics of the fabric forwarding model (indeed, that is the point of having a fabric!). Because the fabric can evolve independently of the edge, multiple forwarding models can exist simultaneously.

3.5 Mapping the Edge Context to the Fabric

The complexity in an edge/fabric architecture lies in mapping the edge context to network addresses or paths. By "mapping" we simply mean figuring out which network address or label to use to given a packet. That is, when a packet crosses from the edge to the fabric, something *in the network* must decide with which fabric-internal network address to associate with the packet. There are two primary mechanisms for this:

Address translation. Address translation provides the mapping by swapping out addresses *in situ*. For example, when the packet crosses from the edge to the network, the edge addresses are replaced with fabric internal addresses, and then these addresses are translated back into appropriate edge addresses at the destination. The downside of this approach is that it unnecessarily couples the edge and network addressing schemes (since they would need to be of the same size, and map one-to-one with each other).

Encapsulation. A far more popular, and we believe more general, approach to mapping an edge address space to the fabric-internal address space is encapsulation. With encapsulation, once a packet crosses from the edge to the network, it is encapsulated with another header that carries the network-level identifiers. On the receiving side, the outer header is removed.

In either case (address translation or encapsulation), a lookup table at the edge must map edge addresses to network addresses to get packets across the fabric. However, unlike the fabric forwarding problem, this lookup may include any of the headers fields that are used by the edge. This is because more sophisticated functions such as filtering, isolation, or policy routing (for example, based on the packet source) must be implemented at the edge.

There are many practical (but well understood) challenges in implementing such a mapping that we will not cover in this paper. These include the control plane, which must maintain the edge level mappings, connectivity fault management across the fabric, and impact of the addressing to basic operations and management.

4 Questions and Implications

Isn't this just another approach to layering? To some extent, one could view the edge and the core as different layers, with the edge layer running "over" the core layer; to that extent this is indeed just another approach to layering. And layering provides some of the same benefits we claim: it decouples the protocols in different layers (thereby increasing innovation) and allows for different layers to have different scopes (which is important for scaling). However, current dataplane layering can be thought of as "vertical", making distinctions based on how close to the hardware a protocol is, and each layer goes all the way to the host. When a layer is exposed to the host, it becomes part of the host-network interface.

What we are proposing here is more of a "horizontal" layering, where the host-network interface occurs only at the edge, and the general packet-switch interface exists only in the core. This is a very different decoupling than provided by traditional layering. We not only want to decouple one layer from another, we want to decouple various pieces of the infrastructure from the edge layer entirely.

What does this mean for OpenFlow? This approach would require an "edge" version of OpenFlow, which is much more general than today's OpenFlow, and a "core" version of OpenFlow which is little more than MPLS-like label-based forwarding. One can think of the current OpenFlow as an unhappy medium between these two extremes: not general enough for the edge, and not simple enough for the core.

One might argue that OpenFlow's lack of generality is appropriately tied to current hardware limitations, and that proposing a more general form of OpenFlow is doomed to fail. But at present much edge forwarding in datacenters is done in software by the host's general-purpose CPU. Moreover, the vast majority of middleboxes are now implemented using general-purpose CPUs, so they too could implement this

edge version of OpenFlow. More generally, any operating system supporting OpenvSwitch or the equivalent could perform the necessary edge processing as dictated by the network controller. Thus, we believe that the edge version of OpenFlow should aggressively adopt the assumption that it will be processed in software, and be designed with that freedom in mind.

Why is simplicity so important? Even if one buys the arguments about the edge *needing* to become more flexible (to accommodate the generality needed in the host-network interface) and *able* to become more flexible (because of software forwarding), this doesn't imply that it is important that the core become simpler. There are two goals to simplicity, reduced cost and vendor-neutrality, and the latter is probably more important than the former. Even if the additional complexity were not a great cost factor, if one is striving for vendor-neutrality then one needs an absolutely minimal set of features. Once one starts adding additional complexity, some vendors will adopt it (seeking a competitive advantage on functionality) and others won't (seeking a competitive advantage on cost), thereby lessening the chance of true vendor-neutrality. We believe this is likely to happen with the emerging OpenFlow specifications, but would not apply to simple label-switching boxes.[5]

What does this mean for networking more generally? If indeed we arrive at a point where the edge processing is done in software and the core in simple hardware, then the entire infrastructure becomes much more evolvable. Consider the change from IPv4 to IPv6; if all IP processing were done at the edge in software, then simple software updates to hosts (and to the relevant controllers) would be sufficient to change over to this new protocols. While we started the paper focusing on infrastructure over architecture, this is one way in which an improved infrastructure would help deal with architectural issues.

Isn't all this obvious? Yes, we think so. But we also think it is important, and is not being sufficiently addressed within the SDN community. We hope that HotSDN will be a forum for a discussion of these topics.

5 References

[1] Beacon: A java-based OpenFlow control platform. `http://www.beaconcontroller.net`.

[2] Brocade VCS Fabric. `http://www.brocade.com/downloads/documents/white_papers/Introducing_Brocade_VCS_WP.pdf`.

[3] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and K. van der Merwe. Design and Implementation of a Routing Control Platform. In *Proc. of NSDI*, April 2005.

[4] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *Proc. of NSDI*, April 2012.

[5] Of course, vendors will always compete in terms of various quantitative measures (*e.g.*, size of TCAM, amount of memory), but the basic interfaces should be vendor-neutral.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *Proc. of SIGCOMM*, August 2007.

[6] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proc. of Usenix Security*, August 2006.

[7] M. de Prycker. *Asynchronous Transfer Mode: Solution for Broadband ISDN*. Ellis Horwood, Upper Saddle River, NJ, USA, 1991.

[8] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a Network Programming Language. In *Proc. of SIGPLAN ICFP*, September 2011.

[9] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM CCR*, 35(5):41–54, 2005.

[10] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM CCR*, 38, July 2008.

[11] Juniper QFabric. `http://www.juniper.net/us/en/dm/datacenter/details/`.

[12] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. of OSDI*, October 2010.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[14] A. K. Nayak, A. Reimers, N. Feamster, and R. J. Clark. Resonance: Dynamic Access Control for Enterprise Networks. In *Proc. of SIGCOMM WREN*, August 2009.

[15] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *Proc. of SIGCOMM*, August 2009.

[16] B. Pfaff, J. Pettit, T. Koponen, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In *Proc. of HotNets*, October 2009.

[17] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent Updates for Software-Defined Networks: Change You Can Believe In! In *Proc. of HotNets*, November 2011.

[18] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, IETF, April 2001.

[19] A. R. Sharafat, S. Das, G. Parulkar, and N. McKeown. MPLS-TE and MPLS VPNs with OpenFlow (Demo). In *Proc. of SIGCOMM*, August 2011.

[20] S. Shenker. The Future of Networking, the Past of Protocols. http://www.youtube.com/watch?v=YHeyuD89n1Y.

[21] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In *Proc. of OSDI*, October 2010.

[22] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. In *Proc. of DANCE*, 2002.

[23] Trema: An Open Source modular framework for developing OpenFlow controllers in Ruby/C. `https://github.com/trema/trema`.

[24] A. Voellmy and P. Hudak. Nettle: Taking the Sting Out of Programming Network Routers. In *Proc. of PADL*, January 2011.

[25] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-based Networking with DIFANE. In *Proc. of SIGCOMM*, August 2010.