Structuring Network Control Plane Computations

Martín Casado[*] Teemu Koponen[*] Murphy McCauley[†‡] Scott Shenker[†‡]

ABSTRACT

Networks have both a control plane and a data plane. Network data planes are built around layers, a modularity that has stood the test of time. However, we have not yet found a similarly sound foundation for network control planes. Instead, control planes are handled through a mixture of manual, fully distributed, and fully centralized mechanisms, none of which provide the needed flexibility (to handle the wide variety of current and future requirements) and scalability (to accommodate ever-increasing network sizes). To fill this void, we propose a new computational model for network control planes that builds on and extends the Software-Defined Networking (SDN) paradigm.

1 Introduction

It is often useful to consider network designs as having two "planes", a control plane and a data plane. In oversimplified terms, the network data plane defines how a router (or switch) uses a packet header and existing forwarding state to make forwarding decisions. The network control plane defines how the forwarding state is computed. Because forwarding state depends on the network as a whole, the network control plane involves an inherently distributed computational task; in contrast, the network data plane is purely local, using state that already exists at the router.

The foundation for the current networking data plane was laid over 30 years ago, with the invention of the Internet and Ethernet (and other L2 technologies). The data plane is highly modular, consisting of multiple layers with relatively clean interfaces between them. This approach has stood the test of time, largely because the demands on the data plane — deliver packets to their intended destination quickly and reliably — have remained largely unchanged.

In contrast, the demands on the network control plane have changed dramatically. In the beginning, the main requirement was just end-to-end connectivity, which was met by fully distributed routing algorithms (involving each router in a distributed computation). Now there are many additional demands on the control plane, including:

- *High-availability:* Networks should be able to withstand isolated link failures without loss of packets.
- *Access control and isolation:* Networks should be able to prevent packets from certain hosts reaching other specified hosts, and maintain traffic isolation between groups of users.
- *Virtualization:* Networks should be able to give tenants their own logical network, which they can configure as they see fit.
- *Traffic engineering:* Networks should manage their traffic so that links do not suffer persistent overloads.

If recent history is any guide, this list is destined to grow over time. Moreover, these demands are being placed on larger and larger networks (*e.g.*, current large datacenter networks connect on the order of 10^5 hosts). Thus, modern control planes must be both flexible (to meet the variety of current and future demands) and scalable (to cope with the ever-increasing network size).

Control planes thus pose a formidable computational challenge; *how do we compute the necessary forwarding state in a manner that is sufficiently flexible and scalable?* To address this question, we first review how control planes are built today. Networks typically use fully distributed routing algorithms to achieve basic connectivity. Some of the additional control plane requirements are met by manual (or scripted) configuration of individual routers, while others are handled in a more centralized fashion; for instance, traffic engineering is often handled by centralized offline computation of MPLS paths, and newer "network fabric" products for datacenters (see, *e.g.*, [2,11]) also use centralized management for some tasks. Thus, the current control plane mechanisms are a mixture of manual configuration, fully centralized management, and fully distributed algorithms. However, none of these computational models can support a sufficiently scalable and flexible control plane, for the following reasons:

[*]Nicira
[†]International Computer Science Institute (ICSI)
[‡]UC Berkeley

- *Manual configuration:* This approach clearly does not scale, nor is it capable of dealing with highly-dynamic environments.

- *Fully distributed:* While these algorithms can often scale for simple problems, such as shortest paths, they are difficult to design and must be specifically tailored to each new requirement. For instance, installing k link-disjoint paths between every two destinations requires a new (and quite complicated) distributed routing algorithm. Since these distributed routing algorithms are delicate, and take years to deploy, this does not provide the necessary flexibility.

- *Fully centralized:* This paradigm, while conceptually easy and fully flexible, clearly does not scale to arbitrary sizes. The failure to scale is along two dimensions: the sheer number of routers/switches (which puts too large a load on a single location), and the geographic scope of the network (the latencies grow too large to centralize all decisions in a single location).[1]

Thus, we are facing a situation where we have a growing list of control plane demands, but do not have a solid computational model with which to meet these demands. This paper outlines an initial attempt to describe such a computational model, one that builds on standard notions of aggregation and hierarchy. Control planes are inherently distributed, so our computational model must cope with distribution. However, the approach need not work for all distributed computations; instead, we leverage the locality of network control plane problems to achieve sufficient scalability and flexibility. Thus, this is not a paper about distributed computing generally defined, which assumes no such locality.

Our approach builds on recent work on Software-Defined Networking (SDN). While the general SDN architecture has been widely discussed,[2] little has been written about how to structure large-scale control plane computations (other than to just centralize the computations). The closest description is in the Onix paper [12], but the distribution models described therein are fully general and are not tailored to specific control plane computations. Moreover, the Onix design is several years old, and we have learned much about SDN systems since then.

In describing our computational model, our goal is to both *clarify* the SDN paradigm — since the current literature has substantial pedagogical gaps — and *extend* the SDN paradigm to include a scalable and flexible computational model. However, when one strips away the networking context and looks at our design as merely a general programming framework, then much of what we say should be completely familiar to the systems community. In fact, our goal is precisely to apply well-known systems techniques to the problem of network control plane computations.

In the next section (Section 2), we describe two basic building blocks of our design: logical servers and logical xBars. We then discuss how to structure the control plane computations in Section 3. In Section 4, we describe how the logical xBars can be built so that they are generally applicable (*i.e.*, not tied to a specific network context) and provide basic consistency properties. We end the paper in Section 5 by examining some of the implications of our approach for SDN (and networking more generally).

2 Building Blocks

In this section we describe the two fundamental building blocks we use in the this computational model: logical servers and logical xBars. Logical servers are the basic unit of computation, where the forwarding tables are actually computed; logical xBars are the basic networking component, which are controlled by these forwarding tables. We end this section by discussing how these fit within the SDN paradigm.

2.1 Logical Servers

The computations for the control plane must be performed on various computational engines in the network. We will model these engines in terms of *Logical Servers*. Each logical server looks like a single machine, but is in fact a group of machines tied together via some distributed mechanisms. There are two separate needs for distribution:

- *Availability:* These logical servers must be highly available, which can be provided with standard availability techniques such as cold/hot-standby mechanisms or state machine replication.

- *Capacity:* If the size of the computation becomes too large for a single machine, then it must be spread over multiple machines (*e.g.*, distributing an all-pairs shortest path computation). The specific distributed algorithm will depend on the particular computation.[3]

In what follows, we will assume that all computations are done on highly-available and high-capacity logical servers. However, in this paper we do not describe the underlying replication and distribution mechanisms used to build these logical servers because in the former case they are standard techniques, and in the latter case they depend in detail on the particular computation (and, in practice, are rarely needed).

[1]However, as we note later, automatic failover measures (as in FRR and the like) greatly reduce the latency issue.

[2]We do not give a comprehensive review of the literature here, but see [10,12,13,15] for the first descriptions of its current formulation, and [3,5,6,9] for some of its intellectual forerunners. There has also been a substantial amount of more recent work (see, *e.g.*, [1, 4, 8, 14, 17, 19–23], but none of it bears directly on the question we address here, which is how to structure large-scale control plane computations.

[3]This form of distribution is typically quite delicate, so in general we advocate structuring the control plane computation so that this is unnecessary (*i.e.*, that no portion of the computation is too large for a single machine, which is typically feasible given the increasing power of highly-multicored computers). We discuss this process of structuring the computation in Section 3.

2.2 Logical xBars

We typically think of physical switches or routers as the fundamental unit on which networks are built; that is, they are typically considered the smallest unit of network functionality and thus are the fundamental entities managed by the control plane. However, rather than being an inseparable whole, a large switch is itself a complicated distributed system. Such switches contain many individual linecards, and each linecard functions as a mini-switch delivering packets between its own ports. These linecards are held together by a chassis that provides high-speed interconnection, and their actions are coordinated by an overall switch control layer. The aggregation of these smaller linecards into a larger switch is so seamless that we typically do not even consider the individual components when thinking about the control plane. Rather than being an oversight, this process of "switch aggregation" — of building larger switching components out of smaller ones, and being able to think about these larger switches without worrying about their constituent components — is fundamental to how we structure the network control plane.

To capture this notion of aggregation, we define a *Logical xBar* as the fundamental unit of switching, and it is these logical xBars that are managed by the network control plane. By using the term "logical" we mean that they are not necessarily a single physical box but instead can be physically distributed, built out of a collection of subsidiary logical xBars. This process can recurse, with these subsidiary logical xBars being comprised of still smaller logical xBars.

These logical xBars are similar to what are now called network "fabrics"; we use a different terms because commercial fabrics are targeted at datacenters, and embody assumptions about full bisection bandwidth and low latency that are appropriate to that context but not generally applicable. However, we do not use this distinction to imply novelty — the notion of fabrics is well established — we only want to remove the association with these limiting assumptions. In another paper [7], we motivate the need for these fabric-like switching structures in SDN on very different grounds, but here we merely investigate how one structures control plane computations using these fabric-like structures.

A logical xBar is essentially a switch, transporting packets between its set of (logical) ports. Thus, its forwarding behavior can be described in terms of a forwarding table[4] comprised of $<$ *header , action* $>$ entries, where the actions are selected from some predefined set that includes forward-out-specified-port(s), drop, modify-header, encapsulate and others.[5] The focus of our paper is on how one organizes the computation of these forwarding entries.

[4]While for convenience we will talk about a single forwarding table per logical xBar, in actual implementations it is often more efficient to use a series of lookup tables. This optimization would not change anything fundamental in what we say below (since any series of tables can be represented by the single table formed by their Cartesian product).

[5]WAN settings probably will require additional QoS-related actions.

These logical xBars not only follow the dictates of the forwarding tables, they also detect local conditions. In the simplest incarnation, these local conditions include only the status of their ports (whether they are up or down), but more sophisticated versions could also describe the estimated latency and available bandwidth between every pair of ports (and other QoS-related measures). This latter issue is obviously more of a concern for WAN deployments, where providing full bisection bandwidth and low-latency is often not possible.

To summarize, the interface to logical xBars should include the ability to both specify the forwarding table and inspect (and set triggers on) local port status and other relevant state.

2.3 Relationship with SDN

As noted earlier, we present our computational model within the context of SDN. What we say here depends not on the specifics of SDN as currently being pursued (*e.g.*, our comments do not depend on the details of OpenFlow or of specific network operating systems), but it does depend on the general approach of using several basic abstractions as the organizing principle for the network control plane. These abstractions are, to some extent, the essence of SDN [18], so in that sense our approach is dependent on SDN.

SDN separates the control plane from the data plane, with the data plane being implemented on switches and the control plane running on one or more network-attached servers. We adopt that approach here, but apply it to our logical building blocks, not just the physical switches and servers. Thus, each logical xBar is associated with a controller running on a logical server.

SDN can operate either in a reactive mode (flow entries are determined only after initial packet is sent to a controller) or proactive mode (flow entries are proactively placed in switches). This paper addresses the proactive case. However, we feel a similar approach would also apply to the reactive case, except that "configuration" would include the control logic needed to make determinations based on the incoming packet rather than a fixed set of flow entries.

3 Structuring the Computation

As noted earlier, fully centralized approaches to control plane computations are flexible but not scalable, while fully distributed approaches are more scalable but less flexible (because the distributed algorithm must be redefined for each computational problem). Here we compromise between these two extremes by leveraging the natural locality of networks.

3.1 Locality + Aggregation = Hierarchy

In structuring our computations, we exploit a property common to almost all networks: locality. We assume two forms of locality:

1. *Connectivity locality:* Links are more likely to exist between nearby network nodes than remote ones. In

contrast to a random graph, where links point randomly to other nodes, in almost all engineered networks the connectivity pattern is far more localized, and links to distant nodes are rare.

2. *Control locality:* Control policies that apply to one network node are more likely to apply to nearby network nodes than to remote ones.

Informally speaking, when you aggregate a linked structure with a high degree of locality, and then look at the connectivity patterns of the resulting larger elements, the degree of connectivity between them is relatively sparse; that is, most of the connectivity is hidden within the aggregates. Repeating this pattern of aggregation builds a hierarchical structure, with each lowest level unit belonging to a series of higher level ancestors, and with a set of (relatively sparse) interconnections between each level of ancestors.

Applying that to our problem here, at the first level of aggregation into logical xBars, we identify regions of the network that are highly interconnected and/or subject to the same control policies; racks in a datacenter and PoPs in a WAN are two such examples. We then aggregate the physical switches in these regions into a single logical xBar. We then look at the network formed by the interconnection between these logical xBars, and repeat the process of looking for regions that can be further aggregated. For a WAN, the levels of aggregation may include: physical switch, PoP, regional network, national network, continental network. At every level the degree of internal connectivity is higher than external connectivity, and the nature of the internal control policies are relatively homogeneous. For a datacenter or general enterprise network, there are probably fewer layers of hierarchy, but the same process would apply. The key is that the locality is preserved at every level of aggregation. For example, the regional networks are not connected in a random graph; rather, neighboring regions are far more likely to have direct connections than remote ones. Of course, this kind of aggregation is standard (and, in particular, is used in approaches such as PNNI [16]), but here we are making the point that we can leverage this kind of aggregation to create a unified computational model for the control plane.[6]

One can then build a computational model on this hierarchical structure, one that combines aspects of centralization and distribution. At each level of the hierarchy there is a local (*i.e.*, centralized) computation, but it is based on information (status updates) from below and directives (forwarding tables) from above. We now describe this more fully.

3.2 General Computational Model

Using the process of aggregation described above, we assume that the logical xBars are arranged in a hierarchy, with each logical xBar being comprised of some number of smaller (*i.e.*, more localized) logical xBars. Similarly, each controller has a logical parent in this hierarchy. There is a connectivity graph at every level in this structure (connecting, for instance, the regional networks), and the forwarding tables at this level of aggregation dictate how to forward between the logical xBars at this level.

Parent controllers in this hierarchy send "configurations" to their children; for now, think of these configurations as a set of static forwarding table entries. In return, children send information about local "status updates" to their parents; for now, think of these as statements about which of their ports are up or down. Thus, configurations travel down the hierarchy, and status updates travel up the hierarchy.

Consider a logical xBar A, made up of a set of subsidiary logical xBars x, y, z. Controller C_A manages the former, and controllers C_x, C_y, C_z manage the latter. Controller C_A receives from its parent controller C_P (which manages parent logical xBar P) a set of forwarding entries, which it must then support. It does so by determining what forwarding entries in each of its subsidiary xBars x, y, z would produce the appropriate forwarding behavior in logical xBar A. It then sends those forwarding entries to the controllers C_x, C_y, C_z managing these subsidiary xBars. This process recurses in the obvious fashion.

In practice, however, the instructions passed from parent to child need not be a static set of forwarding entries, but could instead be a more general specification that dynamically maps a set of local conditions (*e.g.*, which ports are up) into an appropriate set of forwarding entries. These instructions can also dictate which local state changes should be communicated to the parent. For convenience, we will call these "dynamic configurations" to distinguish them from purely static ones (and, in what we write below, the term configuration should be read to include dynamic configurations). These dynamic configurations could, in full generality, be an arbitrary program that takes local conditions as an input and produces local forwarding entries (and status updates) as an output.

This extension to more dynamic configurations allows the logical xBars to do local failover (as in FRR or ECMP) without first having to inform the parent controllers about status updates and then receive instructions about how to respond. This greatly reduces the time it takes to respond to failures. It also can reduce the amount of state information being passed up the hierarchy, though this is typically not the main benefit. Thus, this ability to implement dynamic configurations at the children greatly increases the responsiveness and scalability of the computational model.

We started this section by talking about the role of locality in networks. The ability to do local failover is merely another way increasing locality in the overall design, helping control decisions (and status updates) remain local. Thus, locality is both the basis for our design, and one of its main outcomes.

[6]While we end up with a hierarchical structure, we should make clear that *nothing* in our approach assumes that the network is explicitly hierarchical (though many networks are), nor do we assume that the control plane hierarchy exactly mimics such a network hierarchy (if one exists).

4 Building a Logical xBar

We now discuss how one might build a logical xBar. The goal of a logical xBar's implementation is to achieve forwarding behavior consistent with the forwarding entries given to it by its parent; the most important consideration when designing this implementation is that these logical xBars must be general purpose, usable in any network context. This implies that a logical xBar's internal functioning should be completely decoupled from the protocols used externally (see [7] for a more lengthy version of this argument).

A simple design consistent with this imperative is to use semantic-free labels (as in MPLS) for internal forwarding; by semantic-free we refer to the fact that these labels have no meaning outside of the logical xBar. At the edge of the logical xBar (*i.e.*, on all of its logical ports), packet headers are mapped into labels; this is the only place where the external world gets mapped onto the internal state of logical xBars. The controller for the logical xBar is responsible for both of these aspects:

- Mapping incoming packet headers to internal labels.[7]
- Setting up forwarding entries in the subsidiary logical xBars for these labels so that packets are delivered appropriately from the ingress to the egress of the logical xBar.

This is exactly the MPLS paradigm, which we think should be incorporated into SDN (see [7]).

In the initial SDN designs, where changes were made directly to forwarding entries on each switch, and those forwarding entries referred to the packet's original header (rather than to a semantic-free label), great care had to be taken to ensure that packets were forwarded in a consistent manner (see [17]). To avoid this additional complexity, we design this consistency into the basic operation of these logical xBars as follows.

To make a change in its forwarding, the controller for the logical xBar first inserts a new set of forwarding entries with new (*i.e.*, previously unused) labels into the forwarding tables of the subsidiary logical xBars. Each subsidiary logical xBar must confirm when the new entries are installed. Since the logical structure is recursive, this might require changes to propagate down several levels before the confirmations start traveling up the hierarchy.

Only after these new entries have been inserted into and confirmed by all subsidiary logical xBars does the controller modify the edge mappings (which requires updating forwarding entries in any subsidiary xBar with an external port) to insert these new labels in incoming packets. This guarantees that all packets will be controlled by consistent forwarding state; a packet is either forwarded using the old labels over the entire path (and thus is following the original flow entry), or it is forwarded with the new labels over the entire path (and is thus following the dictates of the new flow entry).

[7]Note that the parent controller specified the mapping between incoming packet headers and the appropriate action; it is the child that determines how to map these headers into labels so that the actions are faithfully implemented.

To clarify this, we return to our previous example with logical xBars P, A, x, y, z. Consider a change initiated by C_P, which sends a new configuration (which we will later call a core configuration; see below) to C_A; this specifies a mapping between external packet headers (which will turn out to be a set of labels, call them outer labels) and actions (for convenience, we assume these actions are to forward out a particular port). C_A then determines what set of configurations (*i.e.*, flow entries) would be needed in logical xBars x, y, z to achieve this behavior, where these configurations include an *edge* configuration — mapping at the edge ports of A between these outer labels and a new set of labels (call them inner labels) — and *core* configurations — mappings at each logical xBar between these inner labels and outgoing ports. The external ports of A would have two actions associated with a packet header; one to insert the inner label (specified by the edge configuration received from A), the other to forward out a particular port (specified by the core configuration). C_A then sends the core configurations to xBars x, y, z. Once it receives confirmation that the core configurations are in place at each xBar, C_A then sends down the edge configurations to those xBars with external ports. Once it receives confirmation that these edge configurations are in place, it confirms to its parent C_P. C_P may then (if logical xBar A has external ports in xBar P) send down an edge configuration mandating how to map from packet headers (which may be labels, or IPv4 header fields) to the outer labels.

We do not have space to discuss partitions in detail, but when a logical xBar becomes internally disconnected (*i.e.*, its subsidiary xBars disconnect into two or more disjoint regions) then (i) if controllers for both segments exist, then they both report to their parent, who the becomes aware of the situation, and (ii) if some xBars are left without a controller then they essentially are no longer part of the network (though this can be managed more subtly, with these elements reverting to some preconfigured default forwarding behavior).

5 Implications

We have described our overall approach to achieving both scalability and flexibility in control plane computations. The scalability follows from the use of aggregation and hierarchy, and the flexibility arises because the control plane calculations at each level of the system are general in nature (and thus can address any of the control requirements we listed in Section 1). We are now preparing an open-source implementation which will provide more concrete evidence for the viability (or not) of our design. Since our remaining space is limited, we briefly review some of the broader implications of our approach.

Clean separation of design concerns. The nature of logical servers, logical xBars, and overall computational structure are cleanly separated. One could change the structure of the problem (from, say, one hierarchy to another) without changing the nature of the logical servers and xBars; similarly, one could change the replication techniques in the logical servers, or the set of actions supported by the logical xBars, without affecting the overall computational structure.

Clean separation of policy concerns. Each logical xBar can exercise its own judgement about how to implement the forwarding table given to it by its parent. Thus, each logical xBar can follow whatever local policies are appropriate (*e.g.*, load balancing across different failure domains, avoiding the most costly links, etc.) without coordinating with its parent or siblings.

Centrality of the logical xBar interface. Note that at all levels of control except the very lowest level (where entries are inserted into hardware switches), the forwarding entries are specified in terms of the xBar interface. Only at the hardware level does OpenFlow itself serve as the interface. Presumably the two sets of interfaces will be similar, but we expect over time that the xBar interface will be the more widely impactful, with the OpenFlow specification being known only to those charged with mapping the xBar actions to hardware forwarding rules.

Bounded depth of computation. Recall that status updates travel up, then configuration changes travel down, and then confirmations travel up. Therefore, in response to a network event, there is a limited set of interactions that take place, whose extent is a function of the depth of the hierarchy. This is in contrast to iterative distributed algorithms that can take much longer to converge (and, during convergence, have inconsistent routing state).

Transactional nature of configuration changes. All packets are handled by consistent forwarding state (*i.e.*, either a packet is handled by old entries, or new entries, but not a mixture of both). This state (handling a particular packet) may be out-of-date (*i.e.*, it has not yet responded to a failure), but it is not inconsistent.

Absence of layering. There is no need for explicit layering between L2 and L3, or more generally between levels in the hierarchy. One can use similar management tools at all levels.

6 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] CAESAR, M., CALDWELL, D., FEAMSTER, N., REXFORD, J., SHAIKH, A., AND VAN DER MERWE, K. Design and Implementation of a Routing Control Platform. In *Proc. of NSDI* (April 2005).

[4] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., AND REXFORD, J. A NICE Way to Test OpenFlow Applications. In *Proc. of NSDI* (April 2012).

[5] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking control of the enterprise. In *Proc. of SIGCOMM* (August 2007).

[6] CASADO, M., GARFINKEL, T., AKELLA, A., FREEDMAN, M. J., BONEH, D., MCKEOWN, N., AND SHENKER, S. SANE: A Protection Architecture for Enterprise Networks. In *Proc. of Usenix Security* (August 2006).

[7] CASADO, M., KOPONEN, T., SHENKER, S., AND TOOTOONCHIAN, A. Fabric: A Retrospective on Evolving SDN. *in submission*, HotSDN 2012.

[8] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: a Network Programming Language. In *Proc. of SIGPLAN ICFP* (September 2011).

[9] GREENBERG, A., HJALMTYSSON, G., MALTZ, D. A., MYERS, A., REXFORD, J., XIE, G., YAN, H., ZHAN, J., AND ZHANG, H. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM CCR 35*, 5 (2005), 41–54.

[10] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. 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] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., HAMA, T., AND SHENKER, S. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. of OSDI* (October 2010).

[13] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR 38*, 2 (2008), 69–74.

[14] NAYAK, A. K., REIMERS, A., FEAMSTER, N., AND CLARK, R. J. Resonance: Dynamic Access Control for Enterprise Networks. In *Proc. of SIGCOMM WREN* (August 2009).

[15] PFAFF, B., PETTIT, J., KOPONEN, T., CASADO, M., AND SHENKER, S. Extending Networking into the Virtualization Layer. In *Proc. of HotNets* (October 2009).

[16] Private Network-Network Interface Specification Version 1.1 (PNNI 1.1), April 2002. ATM Forum.

[17] REITBLATT, M., FOSTER, N., REXFORD, J., AND WALKER, D. Consistent Updates for Software-Defined Networks: Change You Can Believe In! In *Proc. of HotNets* (November 2011).

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

[19] SHERWOOD, R., GIBB, G., YAP, K.-K., APPENZELLER, G., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the Production Network Be the Testbed? In *Proc. of OSDI* (October 2010).

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

[21] VOELLMY, A., AND HUDAK, P. Nettle: Taking the Sting Out of Programming Network Routers. In *Proc. of PADL* (January 2011).

[22] WUNDSAM, A., LEVIN, D., SEETHARAMAN, S., AND FELDMANN, A. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *Proc. of USENIX ATC* (June 2011).

[23] YU, M., REXFORD, J., FREEDMAN, M. J., AND WANG, J. Scalable Flow-based Networking with DIFANE. In *Proc. of SIGCOMM* (August 2010).