Enabling SDN in old school networks with Software-Controlled Routing Protocols

Laurent Vanbever[†] and Stefano Vissicchio[††] ⋆
Princeton University[†], Université catholique de Louvain[††]

Introduction

Software-Defined Networking (SDN) promises to significantly improve network manageability by enabling direct, and centralized control over the network forwarding state via a well-defined Application Programming Interface (API). Fulfilling this promise though is a challenge for network operators as it often requires heavy modifications to their current network architecture, including: *i)* equipment upgrades, as the vast majority of the installed base of network equipments (*e.g.*, routers) do not support SDN protocols; *ii)* new management, monitoring and provisioning systems; but also *iii)* the need for operators training as managing and debugging a SDN network requires essentially completely different skill sets.

In this short paper, we present a lightweight SDN solution that does not require any new network equipment, nor SDN-specific protocols. While it is less expressive than OpenFlow-based SDN, our solution is powerful enough to fulfill complex traffic engineering requirements (*e.g.*, traffic steering through middleboxes, load-balancing) that are hardly achieved in traditional, configuration-based, networks. Moreover, by minimizing SDN inertial factors and simplifying the management interface exposed to network operators, our lightweight SDN model can provide significant incentives to bootstrap the transition towards SDN [5].

The core of our solution consists of a *flexible* and *vendor-independent* API for SDN controllers that works on top of existing networks. Just as OpenFlow enables to program forwarding entries in a SDN switch, our API enables to *program* forwarding entries in a commercial device such as any Cisco or Juniper router. That API can then be used by a SDN controller such as Floodlight or POX to program the forwarding paths to be used across a traditional network, just as in any other SDN.

We argue that *routing protocols* are good candidates to realize such an API, since they: *i)* are standardized as routers must interoperate, even if they are from different vendors; *ii)* have well-defined behaviors (*e.g.*, shortest-path routing); and *iii)* have been used for years by network operators who now have a good mental model of how they behave.

At a high-level, a router running a routing protocol takes routing messages as input and computes forwarding paths as output. A routing protocol can therefore be seen as a *function* from routing messages to forwarding paths. Such functions are standardized and well-known. Given a forwarding entry to be implemented on a node (*i.e.*, the output) and the set of protocols (*i.e.*, functions) running on that node, the runtime implementing our API automatically computes the routing message (*i.e.*, the input) that, when presented to the node, makes it compute the required entry. Doing so, our runtime *inverts* the function implemented by the router.

In addition to enable SDN functionalities in todays' networks, controlling routing protocols provides an effective and powerful way to accommodate incremental transitions towards SDN. Indeed, by using our API along with OpenFlow, a SDN controller can program both legacy devices *and* SDN-enabled equipment. This contrasts with other works on partial SDN deployment (*e.g.*, [2]) that provide SDN functionalities on SDN-enabled devices only.

In the following, we provide more details on how our runtime translates forwarding paths into routing protocols messages. Due to space constraints, we focus on one particular application: centralized traffic engineering which is often used as a motivation to deploy SDN [1]. We also focus on one family of intradomain routing protocols: Link-State Interior Gateway Protocols (LS IGPs), as they are used in the vast majority of the networks and are easy to understand since they rely on shortest-path routing. Our approach though generalizes to other routing protocols (including interdomain protocols such as BGP) and forwarding mechanisms (*e.g.*, MPLS). While relying on routing protocols restricts programmability to destination-based forwarding, our runtime is powerful enough to arbitrarily: *i)* avoid given nodes or links to ensure absence of congestion, *ii)* steer traffic through middleboxes, *iii)* load-balance traffic on multiple path for specific destinations, and *iv)* pre-provision backup paths. We also want to stress that our runtime is more flexible and efficient than simply provisioning static routes. Indeed, by carefully crafting routing messages, our runtime can adapt the forwarding behavior of many devices at once, in a predictable way, without preventing them from converging on their own. In a sense, the runtime only computes the routing input centrally, but the computation of the forwarding paths is still being performed by the routers—*in a distributed manner*.

Enabling SDN-like fine-grained traffic-engineering in old school networks

To program routers forwarding table, our runtime leverages the ability of augmenting the IGP topology with: virtual nodes, virtual links (with arbitrary weights), and virtual destinations (announced by virtual nodes). This virtual topology is computed in such a way that, when presented to the routers, some of them adapt the way they forward the traffic in a given, and predictable way. In particular, augmenting the topology enables the runtime to steer traffic away from any router in the network (*e.g.* by adding a virtual node announcing an existing destination with a small weight) and bring it to any of its neighboring router. To realize this in practice, we assume that our runtime establishes routing adjacencies with all network devices. Such a requirement is easily achieved as most equipments are usually connected to a shared out-of-band management network to simplify management task.

⋆ Alphabetical order.

We now show an example in which our runtime is used to enable per-destination traffic engineering (see Fig. 1). Assume that, starting from the configuration depicted on the left, the link between C and D starts to be congested. To address this issue, the operator (or the SDN controller) may want to partially redirect traffic from C, *e.g.*, sacrificing delay optimality for non-critical traffic towards $D1$ while keeping performance optimality for the red flows directed to $D2$ (cf. middle part). This simple requirement is impossible to achieve in a pure IGP network, because both destinations are attached to D hence the shortest paths from C either *i)* traverses (C, D) for both destinations, or *ii)* do not traverse the link for neither of them.

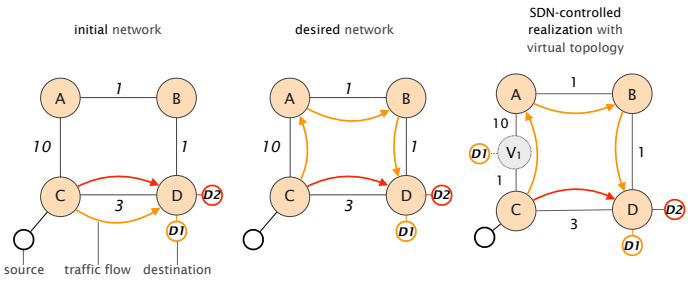

Fig. 1: Lightweight SDN enables fine-grained traffic engineering in existing networks using virtual topology. In this example, it enables to move the traffic sent towards $D1$ by C to A by adding a virtual node v_1.

The right part of Fig. 1 shows how our runtime achieve the desired forwarding behavior, by "faking" the IGP topology presented to the routers. In this case, our runtime pretends that a fake node v_1 is connected to A and C with a weight of 10 (resp. 1) towards A (resp. C), by injecting well-crafted IGP packets [3]. v_1 also advertises that it can reach $D1$. Immediately after the introduction of v_1, C starts to use v_1 to reach $D1$. Indeed, the cost to reach v_1 (1) is smaller than D (3). In practice, v_1 is "mapped" to the link (C, A) in such a way that any packet directed to it from C is sent to A. A then forwards packets to B according to its own shortest-path. Observe that, in this example, C is the only router to change its forwarding path.

While similar results can be obtained today using well-known traffic engineering technologies (*e.g.*, MPLS RSVP-TE), they require end-to-end signaling which can make them extremely slow to converge [1]. In contrast, our solution is centralized and does not require any signaling or device configuration.

Lightweight SDN in practice

Our runtime takes *path requirements* as input and automatically computes the augmented IGP topology. Similarly to [4], our path requirement language is based on regular expressions. For instance, a SDN controller would be able to adapt the forwarding in Fig. 1 by specifying this simple path requirement: $[C, A, *, D1]$. Our path requirement language also supports higher-level operators such as $ECMP(p_1, \ldots, p_n)$ which, given a list of paths $p_1, \ldots, p_n$ with the same source s and destination d, load-balance the traffic on all of them; and $backup(p_a, p_b)$ which automatically provisions p_b as a backup path for p_a in case of failure.

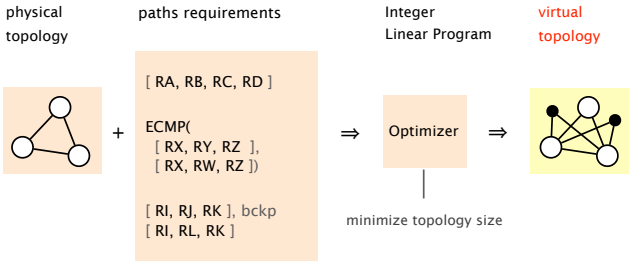

Fig. 2: Lightweight SDN Runtime workflow.

Fig. 2 summarizes the workflow implemented by our runtime system. Starting from the physical topology and the paths requirements, an Integer Linear Program computes the smallest augmented topology that realizes all the requirements. Indeed, as shortest-path algorithms scale with the numbers of links and nodes in the network, we want the minimize the size of topology presented to the routers. While we are using an Integer Linear Program to compute the topology, we also have a polynomial-time algorithm that enables to compute an unoptimized virtual topology. This algorithm could be used after a convergence event such as a link failure to adapt the virtual topology in a timely fashion, while the more costly optimization algorithm can be run at a lower frequency.

Using IGP to encode paths is powerful. Actually, our runtime can make routers forward along *any* set of paths, provided that they are not contradictory. Contradictory paths include: *i)* paths with loops (*e.g.*, $[RA, RB, RA]$) or *ii)* inconsistent paths (*e.g.* $[RA, RB, RC]$ and $[RB, RA, RC]$).

References

1. S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-Deployed Software Dened WAN. In *ACM SIGCOMM*, 2013.
2. D. Levin, M. Canini, S. Schmid, and A. Feldmann. Panopticon: Reaping the Benefits of Partial SDN Deployment in Enterprise Networks. Technical report, TU Berlin / T-Labs, May 2013.
3. G. Nakibly, E. Menahem, A. Waizel, and Y. Elovici. Owning the Routing Table. Part II. Presentation at Black Hat USA 2013.
4. M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. In *HotSDN*, 2013.
5. S. Vissicchio, L. Vanbever, and O. Bonaventure. Opportunities and Research Challenges of Hybrid Software Defined Networks. In *ACM SIGCOMM Computer Communication Review (Editorial Zone)*, 2014.