NOSIX: A Lightweight Portability Layer for the SDN OS

Andreas Wundsam (Big Switch Networks)* Minlan Yu (USC)

1 Motivation: Bridging application expectations and switch diversity

A core promise of SDN is that SDN users are freed from being locked to specific hardware vendors through the use of standardized API's—they should be able to "mix and match" different types and vendors of switches in their network, without having to change the controller application. However, in spite of the standardization of the OpenFlow API, it is very difficult today to write an SDN controller application that is truly *portable*, i.e., that guarantees *correct* packet processing and *good performance* over a wide range of switches.

The reason for this challenge is that the switch landscape is fundamentally diverse: Switches are built in software or hardware, and optimized for different trade-offs of cost and scale. Software switches [1, 4] profit from the fast CPUs and I/O in modern servers on the control plane. On the dataplane however, their performance varies with the load on the server and the characteristics of the installed flows. In contrast, hardware switches [2, 3] only carry a relatively weak CPU, and impose strict resource constraints. Once a flow is installed, however, they provide full bisection dataplane bandwidth. Among hardware switches, there is significant variety in the number of flow tables, the number of flows that can be installed in each table, and the matches and actions supported. It is the authors' belief that such switch diversity is not a short term phenomenon. Consequently, the SDN community has to solve the portability problem.

Fundamentally, the challenge of portability is thus that expectations of the application have to be matched to the feature-set and performance characteristics of the switch that forwards the packets. To bridge this gap, three different approaches have been proposed.

High-Level software abstractions: One approach is to not program the flow model defined by OpenFlow directly, but instead present a higher level abstraction to the application, by e.g., representing the network as a single virtual switch with unconstrained resources [6], or to provide a declarative flow programming language that applications use [5]. A virtual network layer underneath then translates the higher layer application commands to optimized OpenFlow messages, accounting for switch diversity in the process. This approach enables portability and simplifies the programming of applications, but the virtualization layer is highly complex to design and necessarily incurs overhead. Without help, it may be difficult for the virtualization layer to come up with optimal solutions, as flow characteristics are not known in advance. Also, a high level abstraction is less appropriate for applications "close to the metal" that require access to low-level details on the wire.

Simplify the API to the common denominator: The approach taken by the original Open Flow specification was to hide the details by exposing a only core set of features most switches are assumed to provide and ignoring device-specific capabilities. Such an approach makes it easier to write portable programs, but misses the opportunity to improve the applications with useful switch features. For instance, hardware switches often have L2 forwarding tables of significant size (e.g., 25,000 entries), but only support certain operations in them. The more general ACL tables exposed to OpenFlow are comparatively tiny (e.g., 750 entries). At the cost of complexity in the switch, some vendors try to optimize the performance by adaptively mapping one 'virtual' flow table exposed by the switch to several hardware tables, but this requires the application to act in a predictable manner. [1]

Broaden the API to expose all details: Recent OpenFlow versions have shifted towards a bottom-up approach, where network devices expose a rich and detailed model of their internal architecture. Applications can then fully leverage specific device-specific capabilities. However, they also have to manually manage significant complexity: they may have to know how many flow tables are present in a switch, what the control flow is between these tables, and specific requirements for matches and actions. The application programmer, however, may not know ex ante the characteristics of all possible switches, or their performance characteristics.

Neither of the three above approaches are able to solve the diversity problem in the general case. Solving the problem of portability in SDN requires bringing together knowledge from both the application programmer (about what the application is semantically trying to achieve, what its priorities are, what trade-offs it can make) and the switch vendor (about the exact feature set and performance characteristics of the switch). We argue that the right place to do this is in a *lightweight compatibility layer in the controller platform*.

*This majority of this work was performed while at ICSI, funded through a DAAD research fellowship.

[1] Also note that the weak CPUs in many switches prohibit expensive computation.

2 NOSIX : A lightweight compatibility layer in the controller

We introduce a sketch of such a lightweight compatibility layer between the controller applications and the switches, called NOSIX. NOSIX leverages both the application writers' expertise of their requirements and the vendors' expertise of their switch architecture, and provides a unified matching of the two. In NOSIX, application writers group rules and annotate the groups with expectations and optimization goals, while vendors provide software device drivers that map these groups to the hardware, shielding applications from the details they don't care about and allow access to details where they do. This enables more efficient resource usage in the switches, fosters switch innovations, while simplifying controller applications.

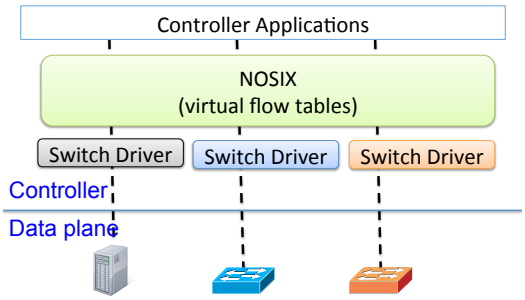

Figure 1: NOSIX architecture

NOSIX separates the application's expectations from the switch-specific implementations, by introducing two key components as shown in Figure 1: (1) A pipeline of multiple **virtual flow tables** to represent application expectations on what rules should be processed at the switch. Each virtual flow table contains flow-based rules with priorities that match packets on different header fields, take actions, and accumulate counters. These flow tables are then connected into the pipeline.[2] (2) The **switch drivers** that provide switch-specific optimizations based on the applications' expectations. Each switch driver is responsible to transform the rules in virtual flow tables into the actual switch flow tables in the forwarding plane. The driver also updates the switch flow table according to the changes of virtual flow tables, and propagates switch events to the applications.

Improve application performance by annotating virtual flow tables: To enable NOSIX to make smart decisions on which rules to install at switches, applications need to expose their requirements and priorities by annotating the virtual flow tables with three properties: *(1) Constraints:* describe application *constraints* that the flow table has to comply with to guarantee good performance of the system. Example requirements include *must-in-hardware*, *minimum_forwarding_rate(500Mbit)*, *minimum_flow_rate(5000 flows/s)*. *(2) Goals:* provide non-binding optimization hints, e.g., given two options, should NOSIX place flows in a table that supports high churn (*goal_churn*), or a table that supports optimal dataplane throughput (*goal_throughput*). *(3) Promises:* To increase the degrees of freedom available to NOSIX in rule placement, applications can give *promises* about their expected utilization of the flow tables. For instance, (a) flows installed for ARP or DNS handling are likely to be short lived and transfer limited traffic. By contrast, (b) flows stemming from the migration of virtual machines are likely to transfer several Gigabytes of data at high rates, but occur only sporadically. The application may increase NOSIX's placement options by declaring a promise of *(a) flow_size(e.g.,* < 10 *pkts/flow)* and *(b) flow_rate (e.g.,* < 5 *flows/s)*.

Separation between rule update semantics and the update mechanisms at switches: When an application modifies the rules in the virtual flow table, the device drivers should update the rules in the switch accordingly. Applications requirements may dictate that such changes are applied with selectable consistency, i.e., with no intermediate states exposed. Switches differ in their support and primitives offered for updates and consistency. We propose to separate the update semantics of applications from update mechanisms at switches.

Summary In summary, NOSIX addresses the portability challenge in SDN controllers by introducing a lightweight portability layer in the controller that acts as a rendevouz point between application knowledge and vendor knowledge. Above, applications specify forwarding rules in virtual flow tables annotated with semantic intents and expectations. Below, vendor specific drivers map them to optimized switch-specific rulesets. NOSIX is designed as a lightweight, "low-level" API that gives applications the possibility to be specific at the cost of portability were necessary. As such, it is well-suited for "bare-metal" applications, and can also serve as a building block on which more high-level abstractions can be built.

References

[1] Cisco Nexus 1000V. `http://www.cisco.com/en/US/products/ps9902/index.html`.

[2] IBM System Networking RackSwitch G8264. `http://www-03.ibm.com/systems/networking/switches/rack/g8264/`.

[3] NEC ProgrammableFlow PF5240 Switch. `http://www.necam.com/SDN/doc.cfm?t=PFlowPF5240Switch`.

[4] OpenVSwitch. `http://openvswitch.org/`.

[5] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *Proc. ACM SIGPLAN ICFP*, 2011.

[6] 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 *USENIX OSDI*, 2010.

[2] The key difference with the pipeline of flow tables defined in OpenFlow 1.1+ is that these virtual flow tables are located at the controller instead of the switches and do not have any resource constraints.