A Slick Control Plane for Network Middleboxes

Bilal Anwer* Theophilus Benson† Nick Feamster* Dave Levin‡ Jennifer Rexford†
*Georgia Tech †Princeton ‡University of Maryland

There is an increasing desire for programs running at a controller to be able to invoke a rich set of matches and actions that span both the control and data planes. Yet, there is no holistic means to do so. While OpenFlow provides a rich, programmable control plane, it has a fixed/narrow data plane; conversely, although middleboxes, a common way to augment the data plane, provide a sophisticated data plane, they do not integrate well with the control plane in that they do not support a similar match/action interface and they do not inform the controller of in-network events.

We envision three broad ways to provide richer matches and actions:

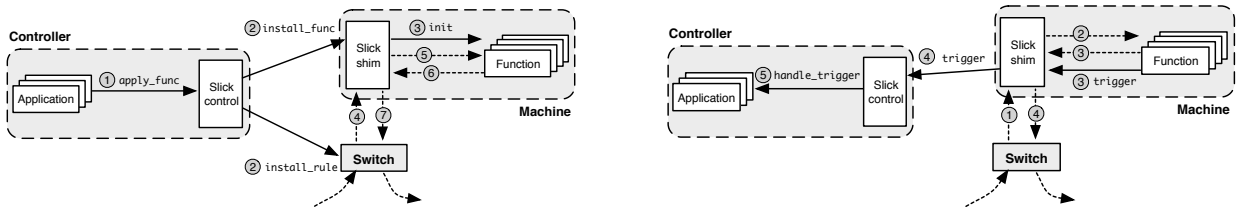
1. *Expand OpenFlow's specification to match on more things and take more actions.* At first blush, this might seem the most logical choice; OpenFlow already provides a small amount of data plane processing (the match-action functions that the switch TCAM provides). However, expanding OpenFlow's narrow data plane processing is an arduous process, requiring standardization and hardware support. Taking it to its logical conclusion, we believe that relying solely on an expansion of OpenFlow's match/action specification will ultimately result in vertically integrated hardware that must be replaced wholesale whenever new functionality is desired—the very fate OpenFlow was arguably designed to avoid.
2. *Have the OpenFlow switch "trap" to the controller whenever a richer match/action is needed.* This, too, seems appealing at first, in that controllers have rich programmatic interfaces, and are not as constrained by hardware as OpenFlow switches. Indeed, this approach has been investigated recently [3]. While this represents a significant step toward more programmable networks, it risks overloading the controller and inflating path lengths.
3. *Put the capabilities to perform richer match/action in software in programmable resources (containing general CPUs, NetFPGAs, NPs, etc.) spread throughout the network, rather than through a structured API that tries to impose a particular data plane model.* In-path data plane programming has seen a recent surge of attention; RouteBricks [1] and CoMb [2], for instance, provide an extensible software data plane. However, they offer only limited support in coordinating among different data plane processing elements, redirecting traffic, or placing/migrating functionality throughout the network.

We argue for the third approach, as it keeps the switches simple and mitigates path inflation. To this end, we introduce *Slick*, a network programming architecture that cleanly decouples processing at the controller and at middleboxes, while also providing an interface by which the two may communicate. Slick does not require expanding the OpenFlow spec because it decouples the data plane of middleboxes, which are complementary to switches. Doing so raises a new set of challenges; for instance, where should the custom data plane functions reside? Slick does not overload the controller, and instead places data plane functions in machines distributed throughout the network. In this manner, data plane processing can occur in-path, similar to systems that provide extensible software data planes.

Slick differentiates itself from prior systems that extend data plane programmability in two major ways. First, Slick can dynamically place sophisticated data plane functionality in the network and steer the right subset of traffic through the right sequence of functions. In so doing, Slick can adapt the placement/replication over time to meet changing network conditions and traffic patterns. Second, Slick allows programmers to coordinate actions between multiple processing entities in the data plane. So doing promotes modularity, reusability, and a consolidation of network resources across multiple applications and policies. The next section describes at a high level the abstractions and separation of concerns that Slick uses to achieve this, but it is worth noting here that this design has allowed us to rapidly develop a wide range of applications.

A brief overview of Slick

A network running Slick consists of the traditional SDN components—a controller and OpenFlow-like routers or switches—as well as what we call *Slick middleboxes*. Slick middleboxes differ from traditional middleboxes in that they are not vertically integrated; rather, one can dynamically load new software onto them. In practice, these can be any programmable computing device, such as a rack server, optionally with an FPGA, NPU or GPU adapter. Network operators must decide into which switches to connect Slick middleboxes, but need not decide a priori what functionality to install.



(a) Installing and running a function.

(b) A packet causing a trigger to be raised.

Figure 1: Slick overview. (a) Packet processing is performed locally at the in-network machine (steps 4–7). (b) Triggers provide asynchronous communication from function to application. (Dashed arrows represent packets.)

A Slick program consists of: (1) Modular *functions* that are distributed throughout Slick middleboxes, and raise asynchronous *triggers* to the controller, and (2) *Applications*, which run at the controller, configures and determines which flows traverse the functions, and handle the triggers they raise. This is how Slick derives its separation of control and data plane functionality: applications can interface directly with the controller, while functions can process packets on the data path. Figure 1 provides an overview of how Slick processes packets in a distributed manner without overloading the controller, and how the data plane functions can raise triggers to inform the control plane.

By default, Slick abstracts much of the underlying network away from the applications and functions. From the application programmer's perspective, the network consists of a controller which is capable of automatically deploying functions in the network and running them on the flows the application requests. From the function programmer's perspective, the network is merely a stream of incoming packets, and a place to throw triggers.

The architecture achieves this abstraction with a "Slick Controller" running at the controller and a "Slick Shim" running on the middlebox. Applications ask the Slick controller to install functions in the network, and to run them on particular sets of flows. The primary jobs of the Slick controller are to dynamically determine where to install or migrate functions, to establish paths so that (only) the correct traffic passes through those functions, and helps deliver messages between applications and functions. The shim is the Slick controller's point of contact in the network; it accepts functions from the controller, runs them, and helps multiplex messages between functions and applications.

Status and open questions

We have implemented the basic components of the Slick architecture, and have found that the clear separation between control plane and data plane processing facilitates rapid development of a wide range of applications. One application that has helped us validate our design consists two functions: the first is a DNS blacklist checker, which performs deep packet inspection on DNS query packets and raises a trigger to the application if it determines that the query is for a blacklisted domain. The application handles this trigger by dynamically requesting the controller to install a second function, which shapes *all* traffic from the host who originated the questionable query.

There remain many exciting open questions regarding the problem of simultaneously choosing where to install (or migrate) functions and via what paths to divert traffic to the functions that should be run on them. Fortunately, Slick places this problem at the controller, and can therefore leverage a global, logically centralized view of the network. Under what conditions should the controller decide to migrate a function and divert existing flows?

Slick currently abstracts away the underlying network from applications and functions, which means that function migration can occur tacitly. We would like to maintain the ability to hide such details, but we believe that applications could benefit from being able to *guide* the controller in its placement. A redundancy suppressing function, for instance, would be most useful if deployed close to the traffic's source. What should the interface be by which applications can specify where their data plane processing components should run, and how should the controller resolve contention among applications? In this sense, we seek a programmatic solution to what network operators currently solve out of band when deciding where into their networks to plug middleboxes.

References

[1] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[2] V. Sekar, N. Egi, S. Ratnasamy, M. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[3] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. FRESCO: Modular composable security services for software-defined networks. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2013.