Sluice: A Network-Wide Model for Programmable Networks

Vikas Natesh Anirudh Sivaraman Ravi Netravali*
NYU UCLA

1 INTRODUCTION

The last several years have seen the emergence of programmable network devices including both programmable switching chips [6?12] and programmable network interface cards (NICs) [13?15]. Along with the rise of x86-based packet processing [16] for middleboxes and virtual switches, these trends point towards a future where the entire network will be programmable. The benefits of such programmability range from commercial use cases such as network virtualization [17] implemented on Open vSwitch [18] to more recent projects that implement packet scheduling [20], measurement [21?23], and application offload of niche applications [24, 25] on programmable switching chips. While the benefits are clear, it is still difficult to program the network as a whole. Current programming languages target individual network devices, e.g., P4 [28] for the Tofino programmable switching chip [6] and the Netronome programmable NIC [13]. However, at present, there is no unified programming model to express and implement general functionality at the level of an entire network, without having to individually program each network device. As an analogy, data scientists prefer using a cluster-wide framework such as Apache Spark or Apache Hadoop that automatically handles scheduling, fault tolerance, and communication between different cluster machines, instead of a framework like MPI that requires the programmer to explicitly program each machine?s computation and communication. Maple [39] was an early example of a network-wide programming model designed for OpenFlow switches. Maple automatically divided functionality between a stateless component running on switches and a stateful component running on the network?s controller. SNAP [40] is a more recent example of network-wide programming; unlike Maple, it additionally offloads stateful functionality to switches by leveraging stateful processing available in several programmable switches. However, both Maple and SNAP cannot express programmable-switch functionality that affects network performance at fine time scales, e.g., packet scheduling [20], congestion control [36], fine-grained measurement of microbursts [23], and load balancing [41]. For these reasons, we have developed Sluice, a programming model that takes a high-level specification of a network program and compiles it into runnable code that can be launched on the programmable devices of network. Sluice endows network operators with the ability to seamlessly design and deploy large network programs for various functions such as scheduling, measurement, and application offloading. We demonstrate Sluice's functionality and simplicity of use via two use examples, both running on a Mininet network of P4-programmable switches: traffic matrix generation for network analysis and a streaming join-filter operation.

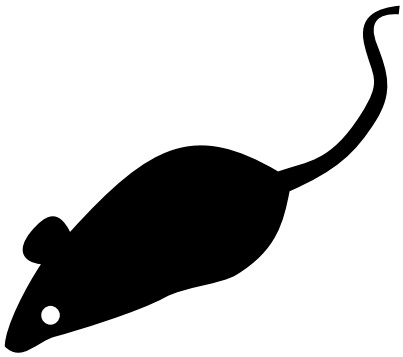

Figure 1: Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

2 SLUICE DESIGN

(Do we want to talk about possible code optimizations? Moving lines around while maintaining intended program flow)

(Do we need to include Sluice BNF grammar?) In the Sluice model, a network-wide program consists of high-level code snippets annotated by the operator to run on particular devices in a network. The code in each snippet is to be executed on packets arriving at its corresponding device. Snippets support a variety of operations: read-from/write-to packets; arithmetic using packet data, local variables, or stateful

*Note

register arrays; control flow statements. To handle computation on custom packet headers not supported by default (ip/tcp/udp/eth), users may define packet declarations similar to C structs. An optional attribute in the declaration, the parser condition, allows the user to restrict snippets to operate on specific flows. Sluice programs may also import device-specific variables/attributes for use in code snippets. Overall, Sluice provides equivalent functionality of P4 but with an ease of use amenable to programming a whole network. The compiler translates a sluice program into per-device programs which are then launched onto the network

3 DEMONSTRATIONS

3.1 Traffic Matrix

Figure 2 displays the Mininet network topology and simulation components. Packets flow over UDP between virtual hosts, through a network of virtual programmable switches. The codelet in Figure 3 is a Sluice program with a single snippet 'traffic_example' that is launched on all switches of the network. To run the simulation, the user passes the Sluice program and network topology to the compiler. The compiler generates P4 code to run on each switch as well as control plane table entries for routing packets through the topology. This demo shows how a very simple Sluice can be used to program each switch to measure link usage and hop counts for a specific flow of user-defined packets on UDP srcPort 1234. Each packet 'p' contains a custom header 'nhops' that will be used by the host to maintain a time-series of hop counts. Specifically, nhops is incremented each time the packet enters a switch, i.e. on each hop. Each switch maintains a stateful register 'counter', indexed by switch ingress port, that tracks how many packets, on the particular UDP flow 1234, have entered through that ingress port. Aggregated over all switches, these counters data represent a matrix measuring each link's usage in the network. This matrix (residing on the whole network) is then queried regularly from the control plane to generate updated traffic time-series plots and statistics.

3.2 Streaming Algorithms

Join-filter example Sluice

4 REFERENCES