Scalable Programming for SDN Controllers

Andreas Voellmy Junchang Wang Y. Richard Yang Bryan Ford Paul Hudak
Yale University

Introduction. A major recent development in computer networking is the notion of *Software-Defined Networking* (SDN), which allows a network to customize its behaviors through centralized policies at a conceptually centralized network controller. In particular, Openflow [4] has made significant progress by establishing (1) flow tables as a standard data-plane abstraction for distributed switches, (2) a protocol for the centralized controller to install forwarding rules and query state at switches, and (3) a protocol for a switch to forward to the controller packets not matching any rules in its switch-local forwarding table. This progress has provided critical components in realizing the vision that a network operator configures a network by writing a simple, centralized network control program, with a global view of network state, thus decoupling the program from the complexities of managing distributed state. We refer to the programming of the centralized controller as *SDN programming*, and a network operator who conducts SDN programming as an *SDN programmer*, or just a programmer.

A critical component in realizing the full benefits of SDN is the SDN programming model, through which a programmer defines its network behaviors. Existing programming models require either explicit or restricted declarative specification of flow patterns, introducing a major source of complexity in SDN programming.

In this project, we explore an SDN programming model that enables an SDN programmer to apply a high-level algorithmic approach to define network-wide forwarding behaviors of network flows. Specifically, the programmer simply defines a function f, expressed in a general-purpose, high-level programming language, which the centralized controller *conceptually* runs on every packet entering the network. When designing the function f, the programmer does not need to adapt to a new programming model but uses a standard programming language to design arbitrary algorithms to classify input packets and return a *network-wide forwarding path* describing how each packet should be forwarded to organize traffic. We refer to this model as *SDN programming of algorithmic policies*. We emphasize that algorithmic policies and declarative policies do not exclude each other. Our system supports both, but this paper focuses on algorithmic policies.

A Motivating Example. To motivate and illustrate the programming of an algorithmic policy, consider a network whose simple policy consists of two parts. First, a security policy: TCP flows with port 22 should use a secure path; otherwise, the default shortest path is used. Using the notion of algorithmic policy, an SDN programmer defines a program `f` that will be conceptually invoked on every packet `pkt`:

```
def f():
  (srcSw,srcPrt) = hostLocation(eth_src())
  (dstSw,dstPrt) = hostLocation(eth_dst())
  if tcp_dst_port_equals(22):
   return securePath(srcSw,srcPrt,dstSw,dstPrt)
  else:
   return shortestPath(srcSw,srcPrt,dstSw,dstPrt)
```

This algorithm, and the program expressing it, are simple and intuitive. The programmer does not think about or introduce switch forwarding table rules—*it is the responsibility of the programming framework to derive those automatically*. In particular, neither the `securePath` nor the `shortestPath` functions specify details such as the match conditions or priorities to use in Openflow rules. Such automation and abstraction provide many additional advantages such as SDN program correctness, performance and stability. For example, the network may have mixed Openflow and traditional routers; or the network may support only source routing. All these will be transparent to the programmer.

In contrast, current mainstream SDN programming models (*e.g.*, NOX [3]) require that programmers explicitly introduce forwarding rules, considering issues such as match granularity, priority levels, and rule overlaps. One might assume that the recent development of declarative SDN programming may help. For example, Frenetic [2] introduces higher-level abstractions including restricted declarative queries and policies as means for programmers to introduce switch-local flow rules. But such approaches require that a programmer extract decision conditions (*e.g.*, conditional and loop conditions) from an algorithm and express them declaratively. This may lead to easier composition and

construction of flow tables, but it still places the burden on the programmer to think along a specific structure, leading to errors, restrictions, and/or redundancies.

Key Challenge. Our approach provides SDN programmers with a simple and flexible conceptual model. But it may come at the expense of performance bottlenecks, if naively implemented. Conceptually, in this model, f is invoked on every packet, leading to a serious computational bottleneck at the controller; that is, the controller may not have sufficient computational capacity to invoke f on every packet. Also, even if one is able to scale up the computational capacity, the bandwidth demand due to every packet going through the controller may not be practical to be satisfied. These bottlenecks are in addition to the extra latency of forwarding all packets to the controller for processing [1].

Rather than giving up the simplicity, flexibility, and expressive power of our high-level programming model, we introduce the design and implementation of novel techniques to address the aforementioned performance challenges. As a result, SDN programmers can enjoy simple, intuitive SDN programming, and at the same time achieve high performance and scalability.

Novel Techniques. Our approach consists of three novel techniques.

- First, we develop a novel SDN *dynamic optimizer* that derives forwarding tables at distributed switches from generic running control programs. Specifically, the optimizer develops a data structure called a *trace tree* that is based on the simple observation that when the programmer-supplied algorithm policy function f is invoked on a specific packet, the outcome can often be generalized and be applicable to a set of packets. As an example, f may have examined only one specific field of the packet and hence the output will be the same for other packets with the same value of the field. A trace tree captures the reusability of previous computations and hence can substantially reduce the number of times that f will be invoked, reducing computational demand, especially when f is expensive.
- The construction of trace trees transforms arbitrary algorithms into a data structure that captures only the essential information and hence leads to our second technique, *policy distribution*, which is a technique for the generation and distribution of switch-local forwarding rules, totally transparent to SDN programmers. By pushing computation to distributed switches, we significantly reduce the load on the controller as well as the latency. In particular, our policy distribution algorithm realizes several novel optimizations, in particular (1) minimization of the number of rules and priority levels in forwarding tables of individual switches; and (2) scalable, network-wide optimization of forwarding tables, instead of traditional algorithms focusing on individual switch forwarding tables.
- Third, we introduce a scalable run-time *scheduler* that complements our optimizer. When flow patterns are inherently non-localized, f may need to be invoked on the central controller many times. Our multi-core SDN scheduler provides substantial horizontal scalability.

Initial Evaluation Results. We have proved the correctness of our key optimizations, developed a complete implementation of all techniques, and evaluated our system through stressing benchmarks. For example, using real HP switches, our dynamic optimizer reduces TCP connection time by a factor of 100 at high load. In addition, our scheduler scales linearly to 40+ cores, achieving a throughput of over 19 million simulated new flow requests per second on a single machine, with 95-percentile latency under 10 ms; At 10 million requests per second rate, the 99-percentile latency is only 0.5 ms.

References

[1] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, 2011.

[2] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[3] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.