Merlin: Programming the Big Switch

Robert Soulé[1], Shrutarshi Basu[2], Robert Kleinberg [2], Emin Gün Sirer[2], and Nate Foster[2]

[1]University of Lugano, souler@usi.ch
[2]Cornell University, {basus,rdk,egs,jnfoster}@cs.cornell.edu

Software-defined networking (SDN) enables programmers to treat an entire network as a single switch that forwards traffic between its outward-facing ports [5]. The "big switch" abstraction presents programmers with a global view of the network that hides the complexities inherent in the physical network, such as distributed state, complicated forwarding rules, and device-specific configuration. But while this abstraction is appealing, an important question remains: *how do we effectively program a big switch?*

Existing SDN programming languages [2, 7, 9, 1, 8] suffer from limitations that make them unable to adequately capture the big switch abstraction—they either focus exclusively on forwarding, or they force programmers to express policies in terms of hop-by-hop functions. Furthermore, none of these languages allow programmers to write policies that specify when richer functions (perhaps implemented using middleboxes) should be applied to packets.

This paper presents Merlin, a new network programming language with three key features that are designed to address the essential aspects of big switch programming: (i) predicates divide multiple classes of traffic over several sub-policies; (ii) path expressions give precise control over forwarding; and (iii) traffic constraints provide bandwidth limits and minimum guarantees. The Merlin compiler maps high-level policies into a constraint problem that statically determines the allocations of network-wide resources such as paths and bandwidth. The Merlin run-time system allows those allocations to be dynamically adjusted, and provides mechanisms for verifying that updated allocations obey the constraints expressed in the original policy.

Merlin Policy Language. As an example to illustrate, consider the following policy, which places an aggregate bandwidth cap on FTP control and data transfer traffic, while providing a bandwidth guarantee to HTTP traffic:

```
[ x : (ipSrc = 192.168.1.1 & ipDst = 192.168.1.3 & tcpDst = 20) -> .* dpi .*
  y : (ipSrc = 192.168.1.1 & ipDst = 192.168.1.3 & tcpDst = 21) -> .*
  z : (ipSrc = 192.168.1.1 & ipDst = 192.168.1.3 & tcpDst = 80) -> .* dpi .*],
  max(x + y,50MB/s) and min(z,100MB/s)
```

It consists of a sequence of statements, followed by a logical formula. Each statement contains a predicate on packet header fields that identifies a set of packets, a regular expressions that describes a set of forwarding paths, and a variable that tracks the amount of bandwidth used by packets processed with the statement. The statement on the first line, with variable `x`, asserts that FTP traffic from host `192.168.1.1` to `192.168.1.3` must travel a path that includes deep-packet inspection (`dpi`). The statements on the next two lines identify and constrain FTP control and HTTP traffic between the same hosts, respectively. Note that the FTP control statement does not include a `dpi` constraint in its forwarding path. The formula on the last line declares a bandwidth cap (`max`) on the FTP traffic, and a bandwidth guarantee (`min`) for the HTTP traffic. Overall, the Merlin language facilitates direct expression of high-level policies, without worrying about how those policies will be enforced in the underlying physical topology.

Network Provisioning. The Merlin compiler implements three tasks: (i) it *localizes* the global policy, or transforms it into one or more locally-enforceable policies; (ii) it determines forwarding paths, places virtual network functions, and makes bandwidth allocations by mapping policies to constraint problems; and (iii) it generates the low-level instructions needed to realize the policy using the switches, middleboxes, and end hosts available in the network.

First, to localize a policy, the compiler rewrites the formula so that each `min`/`max` term constrains traffic at a single location. In other words, given a formula of one term with n identifiers, the compiler produces a new formula of n terms that collectively imply the original. By default, the compiler divides bandwidth equally among the traffic classes, although other schemes are permissible. The `max` term in the example is localized as `max(x, 25MB/s) and max(y,25MB/s)`. Next, the compiler encodes policies into constraint problems that can be solved to determine

forwarding paths through the network, placement of packet-processing functions on individual devices, and allocations of bandwidth. Merlin supports several path-selection heuristics, including one that optimizes for shortest paths and others that minimize the allocation on any link. Although our prototype uses the Gurobi optimizer [3] to find a solution, Merlin is not dependent on any particular algorithm. For example, the abstractions provided by the policy language could be mapped to an approximation algorithm instead. Finally, after the compiler has determined a placement for the system components, it generates the appropriate code to enforce the policy. For forwarding and bandwidth guarantees, Merlin generates instructions for OpenFlow [6] enabled switches and controllers. For middlebox functionality, Merlin generates scripts to install and configure Click [4] modules. Traffic filtering and rate limiting are implemented by generating calls to the standard Linux utilities `iptables` and `tc`.

Dynamic Adaptation. Localizing policies involves an inherent tradeoff: localized enforcement increases scalability, but risks underutilizing resources if the static allocations do not reflect actual usage. Accordingly, Merlin provides a run-time mechanism that allows the static allocations to be dynamically adjusted in a way that respects the administrator-specified global policy. Components called *negotiators* are distributed throughout the network in a hierarchical overlay. Negotiators communicate amongst themselves to dynamically adjust bandwidth allocations to fit particular deployments and traffic loads. They are also responsible for verifying that the new allocations obey the original, non-localized policies. The allocations can be adjusted in two ways: through a central negotiator that acts as a broker, or in a peer-to-peer fashion between participant negotiators. We have implemented negotiators with both min-max fair sharing and additive-increase/multiplicative-decrease allocation schemes.

Experience. We have used our Merlin prototype to specify security policies, declare forwarding paths for campus networks, and ensure bandwidth guarantees for Hadoop jobs in a congested cluster. These examples illustrate how Merlin not only supports a wide range of network functionality including simple forwarding policies, and policies that require rich transformations, but also allows for careful network provisioning and predictable performance through the use of bandwidth caps and guarantees.

Outlook. The success of programmable network platforms has demonstrated the benefits of providing high-level languages for managing networks. Merlin complements these approaches by further raising the level of abstraction, and enabling policies for function virtualization and traffic engineering. The Merlin language and compiler allow administrators to identify classes of traffic; control forwarding paths; and specify traffic constraints. The Merlin runtime provides a flexible infrastructure that allows the network to adapt to changing traffic demands, while respecting the overall global policy. In summary, this approach significantly simplifies network administration, and lays a solid foundation for a wide variety of future research projects on network programmability.

Acknowledgments. Our work is supported by NSF grant CNS-1111698, ONR award N00014-12-1-0757, AFOSR grant FA9550-09-1-0100, a Sloan Research Fellowship, and a Google Research Award.

References

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. In *POPL*, pages 113–126, Jan. 2014.

[2] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, pages 279–291, Sept. 2011.

[3] Gurobi Optimization Inc. The Gurobi optimizer. Available at `http://www.gurobi.com`.

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *TOCS*, 18(3):263–297, Aug. 2000.

[5] J. McCauley, A. Panda, M. Casado, T. Koponen, and S. Shenker. Extending SDN to Large-Scale Networks. In *ONS*, Mar. 2013.

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

[7] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In *NSDI*, pages 1–13, Apr. 2013.

[8] T. Nelson, M. Scheer, A. D. Ferguson, and S. Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *NSDI*, Apr. 2014. To appear.

[9] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*, pages 87–98, Aug. 2013.