Tutorial: Practical Verification of Network Programs

Nate Foster
Cornell University

Arjun Guha
University of Massachusetts, Amherst

Mark Reitblatt
Cornell University

Cole Schlesinger
Princeton University

I. Traditional Networking

Computer networks are essential infrastructure in modern society. Much like the electric power grid, we expect networks to always function, and there are often serious material consequences when they fail. Unfortunately, network failures are all too common. At Amazon, a configuration error during routine maintenance triggered cascading failures that shut down a datacenter and the customer machines hosted there. At GoDaddy, a corrupted routing table disabled their domain name service (DNS) for a day, causing a widespread outage. At United Airlines, a network connectivity issue disabled their reservation system, leading to thousands of flight cancellations and a "ground stop" at their San Francisco hub. Even worse, each of these failures could have been avoided—they were all caused by *operator errors* or *software bugs* [6], [13], [22].

The high rate of network failures should not be surprising. A typical datacenter or enterprise network is a complex system with thousands of devices: routers and switches, web caches and load balancers, monitoring middleboxes and firewalls, and more. Each type of device runs a stack of interrelated protocols and is configured by idiosyncratic, vendor-specific interfaces. Network operators have to grapple with this complexity to implement high-level, end-to-end policies. For example, an access control policy or a quality of service guarantee may need to be implemented by stringing together configurations on several devices. Network operators who can accomplish these feats have been called "masters of complexity" [18], for good reason!

The complexity of traditional networks has also made it extremely difficult to build automated tools for reasoning precisely about end-to-end behavior. To make an effective tool, one would need to somehow reverse-engineer the semantics of numerous poorly-documented devices, construct parsers for proprietary protocols, and formalize their concurrent execution and asynchronous interactions. Although formal models of traditional networks have been developed, they are either too complex to be effective or too abstract to be practical.

II. Software-Defined Networking

Recently, a new network architecture has emerged called *software defined networking* (SDN) that addresses the many of the issues listed above. An SDN eliminates the heterogeneous devices used in traditional networks—switches, routers, load balancers, firewalls, *etc.*—and replaces them with commodity *programmable switches*. These switches are managed and programmed by a logically-centralized *controller* machine, which communicates with switches using a standard protocol such as *OpenFlow* [14].

Since OpenFlow-programmable switches conform to a well-defined interface, it is possible to reason about their behavior and even build formal models of their operation. This has sparked a lot of interest in building verification tools for software defined networks. Before introducing verification, this tutorial will start with begin with an introduc to OpenFlow itself. Using Ox, a simple, OCaml-based controller, participants will first learn how to write some simple SDN applications. The skills they learn will be directly applicable to other popular platforms, such as NOX [7], POX [17], Beacon [3], Nettle [19], and Floodlight [5].

III. Programming with Frenetic

OpenFlow and SDN make network programming *possible*, but they do not make it easy. The first part of the tutorial will make it evident that the OpenFlow abstraction is quite low-level; although it abstracts away several hardware details, it still feels like an "assembly language" for switch programming. It is particularly hard to run several programs or modules simultaneously when programming directly with OpenFlow. If composed naively, two applications are almost certain to destroy each others' network state. Broadly, OpenFlow itself lacks the mechanisms that we need to construct software from separate, modular components.

To address this issue, we will introduce *Frenetic*, a high-level language for programming SDN. Unlike OpenFlow, which requires programmers to carefully manipulate low-level switch-state, Frenetic provides a much higher level of abstraction: a Frenetic program denotes a mathematical, packet-processing function. Frenetic provides a collection of simple functions for filtering, modifying, counting, and forwarding packets, as well as several operators that combine smaller functions into larger ones. The Frenetic compiler takes care of translating these functions into low-level OpenFlow instructions, and the Frenetic runtime system addresses several other details of OpenFlow.

In this tutorial, we will show participants how to program SDNs in a modular way, using Frenetic's abstractions. We will build several realistic network applications from the ground up, and also learn to use more sophisticated modules, such as NAT and MAC-learning, which are part of the Frenetic standard library. We will also look under the hood to see how the Frenetic compiler and runtime system work.

Although the tutorial will focus on Frenetic, we hope to impart an understanding of other *network programming*

ISBN 978-0-9835678-3-7/13.

languages, such as Pyretic [15], Maple [20], and PANE [4]. Although these languages provide a variety of abstractions, they all address issues of modularity and composition that Frenetic also tackles.

IV. Verification with Frenetic

Frenetic's modularity and composition operators make SDN programming much easier; however, SDN promises to make networks verifiable, too. There are several verification tools that operate directly on low-level network state [1], [10], [12], [11], but Frenetic programs can be verified at the source-level.

This tutorial will introduce the Frenetic verification tool, which can check *reachability properties* of source-level Frenetic programs automatically. This tool enables programmers to automatically answer questions such as, "is host A reachable from host B?", "is there a loop involving C?", "is all SSH traffic blocked?", and so on. These are precisely the kinds of questions that network operators ask whilst debugging and troubleshooting their networks.

Under the hood, the Frenetic verification tool operates by encoding programs and properties as SAT formulae and checks their satisfiability using the Z3 theorem prover. Thanks to Frenetic's well-defined, high-level semantics, the encoding is fairly straightforward and certainly much simpler than tools that work with OpenFlow directly.

V. Conclusion

We hope this tutorial will show you how programming languages technology and formal methods can be used to both build networks and verify important network properties. Since this is an in-depth, hands-on tutorial, we will only get to use a small selection of tools and technologies, developed as part of the Frenetic project. However, your experience with Frenetic and its tools will also help you understand the many other languages and tools that have been developed for this domain.

Acknowledgments: Our work is supported in part by the National Science Foundation under grant CNS-1111698, the Office of Naval Research under award N00014-12-1-0757, a Sloan Research Fellowship, and a Google Research Award.

References

[1] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

[2] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *IEEE/ACM Transactions on Networking*, 17(4), Aug 2009.

[3] David Erickson. The Beacon OpenFlow controller. In *HotSDN*, 2013.

[4] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An API for application control of SDNs. In *SIGCOMM*, 2013.

[5] Floodlight OpenFlow Controller. http://floodlight.openflowhub.org/.

[6] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*, 2011.

[7] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR*, 38(3), 2008.

[8] Nikhil Handigol, Srinivasan Seetharaman, Mario Flajslik, Nick McKeown, and Ramesh Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow, Aug 2009. Demo at *ACM SIGCOMM*.

[9] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving energy in data center networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr 2010.

[10] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[11] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[12] Haohui Mai, Ahmed Khurshid, Raghit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[13] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, Yashar Ganjali, and Christophe Diot. Characterization of failures in an operational IP backbone network. *IEEE/ACM Transactions on Networking*, 16(4):749–762, Aug 2008.

[14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[15] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *NSDI*, Apr 2013.

[16] Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A balance of power: Expressive, analyzable controller programming. In *HotSDN*, 2013.

[17] The POX OpenFlow controller, Jul 2011. Available from http://www.noxrepo.org/pox/about-pox.

[18] Scott Shenker, Martin Casado, Teemu Koponen, and Nick McKeown. The future of networking and the past of protocols, Oct 2011. Invited talk at Open Networking Summit.

[19] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, 2011.

[20] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, 2013.

[21] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE), Boston, MA*, Mar 2011.

[22] Zuoning Yin, Matthew Caesar, and Yuanyuan Zhou. Towards understanding bugs in open source router software. In *SIGCOMM CCR*, 2010.