Tutorial: Practical Verification of Network Programs

Nate Foster  Arjun Guha  Mark Reitblatt  Cole Schlesinger
Cornell University  University of Massachusetts, Amherst  Cornell University  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 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 introduction 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
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