Switchyard: A Framework for Network System-Building Projects

JOEL SOMMERS, Colgate University

1 INTRODUCTION

Hands-on activities are critical for learning how network protocols and systems operate. While many instructors choose to have students write sockets-based programs and work at the application layer, others may wish for students to go lower in the protocol stack and implement transport protocols or packet-processing systems such as switches or routers. Carrying out these latter types of projects, however, can pose significant barriers to students because of their technical complexity and difficulties in debugging. They can also impose a great assessment burden on instructional staff.

In this whitepaper, we briefly describe a Python-based framework called Switchyard [1], which can be used for developing network devices in software like switches, firewalls, and routers, as well as to implement transport protocols. Key features of Switchyard include:

- a built-in packet construction and parsing library which allows assembling arbitrary sequences of packet headers;
- integrated testing and debugging facilities, including support for creating test scenarios to help guide system development and debugging, as well as to simplify the assessment process; and
- the ability to run a Switchyard device in a real network environment, which offers the opportunity for students to use off-the-shelf tools (`ping`, `traceroute`, etc.) for diagnostics and for students to connect their devices together, *e.g.*, to create a network of student routers or to connect transport protocol end points implemented by pairs of students.

Exercises and test scenarios are available for interested faculty. Since 2015, Switchyard has been used at 4 institutions (a liberal arts institution and three large universities) in 10 different teaching terms.

Below, we provide an overview of the main features of Switchyard, discuss types of projects that are available for it, outline deployment scenarios, and provide links to resources available online.

2 SWITCHYARD OVERVIEW

For a student, Switchyard provides two main API features: a packet parsing and construction library, and a *network object*. Using the network object, a student can send and receive packets and query available interfaces. The main entrypoint for a program that implements the logic for a network device is a `main` function that accepts one parameter, the network object. Beyond that, a student can structure their program in any way.

Integrated testing and debugging is a primary feature of Switchyard, and its design has been shaped through a focus on these critical tasks. In particular, the network object may represent a collection of real network interfaces on which packets can be sent and received, or it may represent a *fake* network for testing. From the student's (device implementer's) standpoint, there is no difference in how the network object is used. For real mode, Switchyard uses the widely available libpcap to facilitate send and receive operations.

Author's address: Joel Sommers, Colgate University, 13 Oak Drive, Hamilton, New York, 13346, jsommers@colgate.edu.

Test scenarios can be created by instructional staff and distributed to students, or students can be required or encouraged to create their own. If instructor-created tests are distributed to students, it is possible to provide them in a format that does not make it possible for students to reverse engineer high-level code for solving key packet-construction problems [1].

Test scenarios consist of a definition of an imaginary device with a set of network interfaces. Each interface minimally has an Ethernet address, and may have any number of IPv4 and/or IPv6 addresses. The remainder of a test scenario consists of *expectations*, which may either be a packet *input* event (*i.e.*, an expectation that a packet should be received by the imaginary device on one of its interfaces), a packet *output* event (*i.e.*, an expectation that a packet should be emitted by the imaginary device out one or more of its interfaces), or an *input timeout*. As a simple example, we might wish to test for the fact that when a simple Ethernet hub receives a packet on one interface, it should then forward the same packet out all its interfaces except the one on which the packet was received. For packet output events, it is possible to selectively match whether a packet emitted satisfies certain criteria.

When Switchyard is run in test mode, it executes a user-specified program in the context of the test scenario and its fake network, and displays indications of whether tests pass or fail. Passing tests are shown in green, the failing test (if any) is shown in red, and tests that could not be executed because of a prior failure are shown in yellow [2]. When a test fails, a debugging session is started, by default, so that the code and program state can be inspected to try to understand what led to the failure.

It is also possible for a student to explicitly include a function call to `debugger()` anywhere in their program to invoke an interactive debugging session at that point in code execution. For example, a student may want to interactively inspect incoming packets through a debugger session while running their device in a live network in order to evaluate program state and to step line-by-line through their code to debug.

In our experience, the testing capabilities of Switchyard are a huge benefit to both students and instructional staff. For students, explicit tests take away (at least some of) the mystery in networking code (*e.g.*, did my program send a packet? did it receive a response?). Moreover, the immediacy of being put into a debugging session to inspect the context surrounding a test failure can help to illuminate the *why* behind a failure (*e.g.*, was the packet formatted correctly? was it sent out the right interface and with the correct addressing?). For instructors, (partial) tests can be constructed and provided as scaffolding to help guide students toward success, and additional (private and complete) tests can be used for assessment. Even with small classes this can relieve a significant grading burden.

As noted above, Switchyard has a built-in library for packet construction and parsing. The library allows arbitrary sequences of headers to be assembled into a packet. It includes packet header classes for Ethernet, IPv4 and IPv6, along with the typical transport protocols ICMP/ICMPv6, TCP, and UDP. The library is extensible, so that new packet formats can be created, *e.g.*, for implementing a custom routing protocol.

3 PROJECT EXAMPLES

There are a variety of exercises that are available to use as-is, or with modifications [3]. A private instructor repository contains test scenarios and other instructor-only tools. Below, we outline the available exercises:

[1] In essence, the test descriptions are compiled to a bytecode format.
[2] Colored output can also be disabled for accessibility.
[3] Exercises are available on Github.

Learning switch. This exercise involves building a simple Ethernet learning switch. Possible extensions include a modified spanning-tree protocol.
An advantage of using Switchyard for this kind of exercise is that students can design the protocol and wire-format messages for establishing and maintaining the spanning tree, as well as to write tests for protocol correctness.

IPv4 router. This exercise is fairly complex and can be scaffolded with multiple stages, such as ARP handling, longest-prefix match lookup and forwarding (with a static forwarding table), and ICMP handling. The example IPv4 router exercises follow this breakdown of milestones. Extensions include link-state or distance-vector dynamic routing protocols.
Switchyard's debugging and testing facilities are especially useful for this series of exercises, given the complexity of different subtasks. For example, in developing code for ARP processing, a commonly observed approach is for students to preemptively add a call to `debugger()` in order to inspect the contents of the ARP table and other router state when ARP requests arrive.

IPv6 router. This exercise is in progress and will include ICMPv6 router advertisement and solicitation, neighbor discovery, etc., with a static forwarding table. The link-state and distance-vector dynamic routing exercises for the IPv4 router will eventually be adapted for use with the IPv6 router as well.

Firewall and/or DPI middlebox. This exercise involves implementing simple static firewall rules and rate limitation on particular flows. An example extension is to implement some other type of impairment (*e.g.*, probabilistically reset flows toward a particular destination). Another extension to this exercise—or a different exercise altogether—could be to implement a deep-packet inspection middlebox that implements some form of content detection and rewriting. Such exercises can provide good context for discussion on surveillance, censorship, and related ethical issues.

Transport protocol implementation. Switchyard can also be used to implement a transport protocol "underneath" a nearly unmodified Python socket-based program [4]. In a past networking course offering, the author had students create a UDP-based socket application (without Switchyard) as the first assignment; at the end of the course, students implemented a reliable transport within Switchyard, using their (mostly-the-same) program from the first weeks of the semester.

4 DEPLOYMENT SCENARIOS

Switchyard has fairly minimal dependencies (a small number of Python packages, libpcap, and libffi) and is easy to install. In the past, we have used Switchyard in conjunction with Mininet. While the test scenarios are helpful for debugging, seeing one's code run in a "live" environment is exciting and motivating. In Mininet, a student can run their Switchyard device on one or more nodes in a topology and use additional tools, *e.g.*, `traceroute` or `ping`, for diagnosis. Deployment to a VM in a public cloud or in a research and education cloud such as CloudLab is therefore straightforward.

It is also possible to facilitate student collaboration and social learning by having them connect devices together. For example, students might run their routers that implement a dynamic routing protocol on different nodes in a (physical or virtual) topology and verify that they interoperate correctly, or pairs of students could work together to ensure that their transport protocol implementations work correctly in concert. Logistically, student devices could run within the same Mininet

[4]The one difference is that Switchyard's socket emulation library needs to be imported instead of the standard Python socket library.

topology, or could run on separate hosts within a topology running on CloudLab (*e.g.*, using just the experiment interfaces available).

Switchyard can also be run on a commodity operating system, such as macOS or Windows. This type of deployment can be useful in packet sniffing-type exercises, in which students might programmatically analyze packets or implement a passive or active measurement tool. Although Switchyard's performance is not high (thanks largely to Python), the barrier to implementation and experimentation is quite low.

LINKS TO RESOURCES

Faculty interested in using Switchyard can find links to resources below:

- Source code on Github: https://github.com/jsommers/switchyard.
- Documentation: https://jsommers.github.io/switchyard.
- Exercises are available on Github (a separate, private, instructor repository contains premade test scenarios).
- Switchyard can be installed through PyPi: `pip3 install switchyard` (libffi and libpcap must be installed separately; see the README on Github).
- A Docker container is also available (Ubuntu 18.04 with Switchyard): https://hub.docker.com/jsommers/switchyard.

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers who provided helpful feedback for improving this whitepaper. Thanks also to students and faculty who have given suggestions for improvements in both Switchyard's functionality and feature set.

This work has been supported by the National Science Foundation under grants CNS-1814537 and CNS-1054985. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] J. Sommers. Lowering the Barrier to Systems-level Networking Projects. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 651–656, 2015.