Lowering the Barrier to Systems-level Networking Projects

Joel Sommers
Colgate University
jsommers@colgate.edu

ABSTRACT

Developing systems-level networking software to implement switches, routers, and middleboxes is challenging, rewarding, and arguably an essential component for developing a deep understanding of modern computer networks. Unfortunately, existing techniques for building networked system software use low-level and error-prone tools and languages, making this task inaccessible for many undergraduates. Moreover, working at such a low-level of abstraction complicates debugging and testing and can make assessment difficult for instructors and TAs.

We describe a Python-based environment called Switchyard that is designed to facilitate student projects for building and testing software-based network devices like switches, routers, and middleboxes. Switchyard exposes a networking abstraction similar to a *raw socket*, which allows a developer to receive and send Ethernet frames on specific network ports, and provides a set of classes to simplify parsing and construction of packets and packet headers. Systems-level software created using Switchyard can be deployed on a standard Linux host or in an emulated environment like Mininet. Perhaps most importantly, Switchyard provides facilities for test-driven development by transparently allowing the underlying network to be replaced with a test harness that is specifically designed to help students through the development and debugging process. We describe experiences with using Switchyard in an undergraduate networking course in which students created an Ethernet learning switch, a fully functional IPv4 router, a firewall with rate limiter, and a deep-packet inspection middlebox device.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education; C.2.5 [**Local and Wide-Area Networks**]: Internet (*e.g.*, TCP/IP); C.2.6 [**Internetworking**]: Routers

General Terms

Design, Experimentation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCSE'15, March 4–7, 2015, Kansas City, MO, USA.
Copyright © ACM 978-1-4503-2966-8/15/03 ...$15.00.
http://dx.doi.org/10.1145/2676723.2677211.

Keywords

Routing, switching, middleboxes, test-driven development

1. INTRODUCTION

One of the most effective ways to *learn* is to *do*. In computer science, the process of building software and "getting their hands dirty" [7] often leads students to develop deeper understanding. Creating significant software artifacts—especially programs perceived as *real*—can be hugely motivating, engaging, and rewarding, and can spur new ways of thinking about what may otherwise be abstract or complex concepts.

Particularly in computer networking, there have been many systems and tools developed over the past decade to facilitate hands-on activities in order to bring concepts to life. For example, publicly accessible large-scale emulation platforms [10] can be used to experiment with networking protocols and programs, virtualization tools are available for creating and working with complete "networks-in-a-box" [9, 16], and recent generations of simulation tools can seamlessly incorporate "real" network components [3]. Moreover, advanced courses have been created with the explicit goal of building significant systems such as a complete IPv4 router [5]. Each of these platforms represent significant advances to teaching and learning in computer networking.

Unfortunately, many of the most compelling and realistic hands-on environments for building networked systems have a very steep learning curve and present a high barrier to entry for many undergraduates. One reason is that the more realistic systems tend to be implemented using low-level languages and tools, *e.g.*, C/C++. As a consequence, it can be extremely challenging for students to test and debug their programs since the underlying tools are implemented at a rather low level of abstraction. For networked systems, testing often involves the need to create carefully crafted packets in order to verify an intended behavior or feature. As a result, creating and executing tests often involves low-level and error-prone tools, too. Any error or erroneous assumption can be difficult to detect, and can lead to frustration and discouragement for students.

In this paper, we describe a new framework for building real networked system software called *Switchyard*[1]. Switchyard provides a Python API, and helps students avoid pitfalls associated with low-level implementation languages by providing higher-level abstractions with which to work. Our motivation for providing carefully designed abstractions for students to work with is to enable them to focus on the key problems and concepts of building a router, switch, middlebox, or other device, as opposed to struggling with an arcane low-level language environment. Perhaps most importantly, Switchyard provides facilities for *test-driven development*,

[1] `https://github.com/jsommers/switchyard`

so that students can create or be provided with a suite of tests for testing, debugging, and verifying intended behavior. The Switchyard framework enables seamless execution of student programs in either the test harness environment, on a bare Linux host, or in a Linux-based virtual machine environment such as Mininet [9]. In short, Switchyard *lowers the barrier* to system-level networking projects by *raising the level of abstraction* of the tools used to construct networked systems, and by including *purpose-built debugging and testing capabilities* to help students with what are often the most challenging aspects of systems software development.

We describe our initial experiences with using Switchyard in an introductory undergraduate course in computer networking. In this course, students were able to build a variety of systems, including an Ethernet learning switch, a fully-functional IPv4 router, a firewall with rate-limiting capabilities, and a deep-packet inspection middlebox device. Nearly all students completed the various milestones for these projects, which is significant considering the technical and conceptual complexity involved with each system. We discuss observations from student evaluations, and describe our ongoing efforts to expand and improve Switchyard.

2. RELATED WORK

Over the last decade, a number of hands-on educational environments that use either real or emulated network components have been developed. At one end of the spectrum are platforms employing real networking hardware and components. For example, the NetFPGA platform is a compelling platform for developing both software and hardware components of networked systems such as switches and routers [12], and entire courses have been developed with the goal of building a complete IPv4 router [2]. Similarly, the Open Networking Laboratory [20, 21] enables students to remotely access and control real networking hardware in a laboratory setting. Likewise, Yuan *et al.* [6] describe a set of hands-on experiences for students using commodity PCs running Linux, and other free and open source software such as Wireshark, Snort, and the Zebra router. Yoo *et al.* [17] also describe an environment that relies on the Zebra router and a set of scripts to configure hosts in particular logical topologies for use in experiments. Similar to these approaches is the one described by Pan [15] that centers on using the Linksys WRT54GL and the OpenWRT firmware to provide students with a realistic platform for experiments. Lastly, the Wisconsin Advanced Internet Laboratory [4] employs a variety of networking components such as switches and routers, and uses configuration and experiment management software based on Emulab [10].

Hands-on settings that make use of the Emulab software have seen fairly wide use in networking education. These environments allow students to create virtual topologies using a set of commodity hosts, and to emulate different link characteristics such as delay and loss. They enable a fair amount of realism, while allowing lab resources to be multiplexed among students. Laverell *et al.* [11] describe their experiences setting up and using the Emulab software in networking courses. The Tinkernet system described in [13] contains functionality similar to that in Emulab. The authors of that paper describe a set of laboratory experiences that focus on building a networking protocol stack.

While Emulab-type systems virtualize links, other systems present entire virtual network topologies to students, enabling different levels of interaction with the virtual network. For example, the GINI system [14] creates virtual networks using tunnels and Linux virtual machines, enabling students to use standard tools and APIs to interact with the virtual network. Similarly, the Virtual Distributed Ethernet system [8] emulates an Ethernet switch and allows virtual and real machines to be stitched together into topologies that students can interact with using standard tools.

In a similar vein, the Virtual Network System (VNS) [5] allows students to construct virtual network topologies and to safely interact with their virtual network, as well as the rest of the Internet, to send and receive raw packets. As a result, students can design and develop a simplified Internet router that runs in user space. VNS has been decommissioned in favor of the lightweight virtual machine-based Mininet system [9] that provides a Python-based API for creating virtual topologies within a single host. Although Mininet is largely designed for use with Openflow and software-defined networking, it is more generally useful for student experimentation, and projects created originally for VNS have been adapted for use with Mininet [3]. The Netkit system [16] is also based on using Linux virtual machines for building virtual topologies and running networking experiments. A fairly large number of ready-made laboratory exercises have been developed for use with Netkit. It is important to note that for each of these Linux VM-based systems, the programming API is C-based, and there are only limited capabilities provided for testing and debugging (*e.g.*, to help with `printf`-style debugging), unlike Switchyard.

3. SWITCHYARD GOALS AND DESIGN

In this section we discuss the goals and motivation behind Switchyard, and describe its design, implementation, and how it is used in student projects.

3.1 Goals

As discussed above, many of the educational platforms and tools developed over the past several years to enable students to gain hands-on experience with networking protocols and systems software development require use of low-level tools and language environments. Although low-level implementation languages such as C are important for undergraduates to be exposed to, they can make it difficult for students to focus on and grapple with the underlying systems concepts that they *should* be learning. A key motivation for developing Switchyard was to address the limited availability of systems-building environments that leverage higher-level abstractions and tools. Below, we describe the main pedagogical goals of Switchyard.

1. Engage students in building networked systems and protocols, such as Ethernet learning switches, IPv4 routers, and implementations of TCP. As mentioned previously, this is one of the key motivations for Switchyard. We strongly believe that "hands-on" projects are the best ways to develop a deep understanding of networks and other computer systems.

2. Construct the framework such that, as far as possible, students are able to focus on networking and systems concepts rather than on arcane language or API details. Simply put, use of low-level languages can pose a significant barrier for students with limited experience (*i.e.*, many undergraduates) to engage in systems-level projects. Our experience is that raising the level of abstraction through use of a higher-level language can help significantly. Moreover, careful design of the API that students use can help to focus student efforts on core concepts rather than extraneous details.

3. Provide debugging and testing facilities to assist students with development of complex systems software. When facing

[2] `http://www.cl.cam.ac.uk/teaching/1415/P33/`

[3] `https://github.com/mininet/mininet/wiki/Teaching-and-Learning-with-Mininet`

challenging programming assignments, students often do not know where to start. Encouraging a *test-driven development* cycle can help to funnel students through specific functionality milestones toward completing an assignment. Moreover, we believe that in an educational context, it is important to integrate testing facilities with capabilities to assist with debugging and inspection of program state so that when tests fail students have enough context to understand *why* a test fails.

4. Ease the burden of assessing technically complex student projects. As many faculty can attest, examining complex software systems created by students ~~can be~~ is extremely time consuming. Especially for faculty who do not have graduate student TAs to assist with grading, the difficulty posed by assessing such projects creates a disincentive to assigning challenging projects such as construction of an IPv4 router.

3.2 Framework design and API

Switchyard is designed as a Python program and module that exposes a fairly minimal and simple API to a programmer. The conceptual structure of Switchyard is depicted in Figure 1. The API is designed to provide the most basic and essential functions for implementing the core logic of a network device, including sending a packet, receiving a packet, and obtaining a list of ports (interfaces) and their configurations on the device. Switchyard also includes a library to simplify packet parsing and construction.

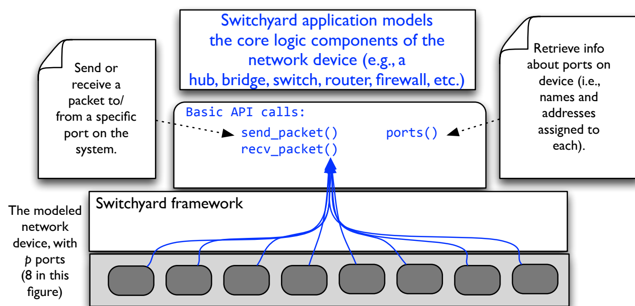

Figure 1: Switchyard design

A program designed to work within Switchyard is written as a Python program that includes one required function: `srpy_main` [4], as shown in Listing 1. This required function must take one parameter, which is an object on which a student can call the core Switchyard API functions `send_packet`, `recv_packet`, and `ports`. As Listing 1 shows, a conventional structure to a Switchyard program is to have an infinite `while` loop within the main entrypoint function. Inside that loop packets are received and handled in some way, *e.g.*, to forward them out one or more ports. The `recv_packet` function returns a tuple containing a reference to a packet and the name of the port on which the packet arrived (*e.g.*, `eth0`). The `recv_packet` can optionally take a timeout value. If no packets arrive before the timeout value, a `Timeout` exception is raised. When the underlying framework shuts down, it raises an exception that can be handled in the device implementation code in order to perform a graceful shutdown.

3.3 Framework implementation

Switchyard is implemented as a Python module that consists of classes to enable interaction with *real* network interfaces on the

[4]The name `srpy` is a reference to the venerable `sr` (software router) projects developed as part of the VNS project [5].

```
def srpy_main(net):
    while True:
        try:
            packet, port_name = net.recv_packet()
        except Shutdown:
            # we got shutdown signal
            break
        # handle an incoming packet
```

Listing 1: Basic structure of a Switchyard program.

host system or to enable interaction with a test harness that involves no actual network I/O. Invoking the framework is accomplished through a command-line interface. The main Switchyard program (`srpy`) requires at least one argument, which is the name of a file containing an entrypoint to the student's network device implementation. The `srpy` program loads the student's file and locates the `srpy_main` function to be executed. Additional command line options determine whether Switchyard starts up in *real* mode or in *test* mode. Each of these modes are described below.

Students can build and parse packets using a library built into Switchyard. In the version of Switchyard that we initially used in the classroom, we delegated calls to the packet parsing libraries in the POX Openflow controller [2]. There were various pitfalls we encountered with that approach, and we developed a new set of packet parsing libraries from scratch (though based on the POX libraries) which we describe in Section 4.2.

3.3.1 Real network mode

By default, `srpy` attempts to use all network interfaces on the host system (except for the loopback interface) for sending and receiving packets. It uses a Python wrapper around `libpcap` for sending and receiving packets, which enables some degree of portability for Switchyard.

Upon framework startup, a thread is started to handle I/O for each separate network interface used. If the underlying libpcap implementation supports it, the threads open pcap devices for each interface in a non-blocking manner. Packets that arrive on any network device are converted to a packet object representation (rather than as a raw sequence of bytes) and added to the tail of a shared queue along with the name of the device on which the packet was received, and a timestamp. When user-level code calls `recv_packet`, the packet at the head of the queue is removed and returned.

When user code calls `send_packet`, it supplies a packet object and the name of the device on which to send the packet as parameters. The Switchyard framework serializes the packet and delivers it to the thread that handles I/O for the given device via another queue. If there are errors associated with the packet object given to `send_packet` or if there is an invalid device name specified (*e.g.*, no device exists for the given name), an error is returned.

One key implication of using libpcap as the low-level network access mechanism is that it is often necessary to install packet filters to prevent host processing of incoming packets. For example, if a Switchyard user program should be responsible for ARP requests and responses, the host system must be prevented from responding. The way this situation is currently handled is that students are given a wrapper script to start Switchyard that installs `iptables` and/or `ebtables` rules. This same pattern of providing custom handling for packets and preventing host processing for the same packets by installing firewall rules is a common approach taken with some network measurement tools, *e.g.*, [18, 19].

3.3.2 *Test harness mode*

When started in *testing* mode, Switchyard requires at least one *test scenario* to be supplied as a command-line argument. The scenario specification defines some set of physical or logical ports that are configured on an imaginary network device, as well as any addressing information associated with each port, *i.e.*, an Ethernet address and (optionally) an IPv4 address and subnet mask. The scenario also defines a series of test *expectations*. In particular, an expectation may be that a specific packet arrives on a given interface of the device or that the Switchyard user program sends a packet out a particular interface.

An (incomplete) example of a program for creating a test scenario is shown in Listing 2. The `TestScenario` class in Switchyard encapsulates various details associated with a set of tests. After constructing the scenario object, three network interfaces with specific Ethernet addresses are added to an imaginary device. Following that, a `PacketInputEvent` expectation is created to indicate that a packet should arrive on a particular interface. The meaning of this statement is that the creator of the test expects the Switchyard user program to call `recv_packet`, and that the packet object `testpkt` will be returned in response to that call. Lastly, a `PacketOutputEvent` expectation is created to indicate that the packet `testpkt` should be sent out two interfaces. The meaning of this expection is that the Switchyard test framework will require the user program to send a packet that *exactly matches* `testpkt` out the two interfaces `eth1` and `eth2`.

```
scenario = TestScenario('hub tests')
scenario.add_interface('eth0', '10:00:00:00:00:01')
scenario.add_interface('eth1', '10:00:00:00:00:02')
scenario.add_interface('eth2', '10:00:00:00:00:03')
testpkt = ... # code to create testpkt packet object
scenario.expect(
  PacketInputEvent('eth1', testpkt),
  '''An Ethernet frame with a broadcast
  destination address should arrive on eth1''')
scenario.expect(
  PacketOutputEvent('eth0', testpkt,
                    'eth2', testpkt),
  '''The Ethernet frame with a broadcast destination
  address should be forwarded out eth0 and eth2''')
```

Listing 2: Test scenario example.

Expectations created for a `PacketInputEvent` are straightforward: the test creator must construct a packet object that will be delivered to the user program under test. While this portion of the example is incomplete, the steps used to create the test packets are identical to the steps required in a standard user program.

Expectations created for a `PacketOutputEvent` are somewhat more complex, because they require *matching* a packet sent by a user program against a reference packet. Currently, Switchyard accommodates exact match testing as well as matching on selected header fields (including support for "wildcards"). The header fields used for this latter type of match are currently the same fields allowed for matching packet header fields in the Openflow 1.0 specification [1]. In addition, arbitrary predicate functions may be supplied, *e.g.*, to test that an IP TTL is decremented by 1 when forwarding an IP packet. Finally, test expectations can be created to handle the situation in which a test creator expects the user program to call `recv_packet`, but the test creator wants the call to `recv_packet` to *time out* and not return anything. This expectation is useful for testing situations in which a protocol implementation must handle the lack of response to a request, such as a failed ARP query.

The test scenario API is designed to be used either by an instructor or TA or by students. In the case of instructor-created test scenarios, it would clearly be inappropriate to distribute the test scenario construction code itself, since that code would naturally contain expressions for creating the same packets that students must construct as part of a given assignment. Switchyard contains facilities to *serialize* a test scenario for public distribution to students. The serialization procedure first uses Python facilities for serializing the `TestScenario` object created in a given test scenario file. The result of this step is a string that contains representations of each of the expectations in a given scenario, including string representations of each of the packets used. Importantly, the program code that created the packet objects is no longer present. Next, the serialization procedure compresses the serialized test scenario, converts it to a base-64 string, and writes that string to a file. This file can be posted as part of an assignment for students to then use while developing their code.

3.4 Example: a simple hub

We now show through an example how a test scenario can be used, and how it can facilitate test-driven development by students. The example user application we use is a network hub—an extremely simple network device that floods any packet received on an interface out all interfaces except the input interface. The test scenario example above (Listing 2) shows the beginning of the testing code used for this example, which consists of 8 total expectations (4 packet input event expectations, and 4 corresponding packet output expectations).

We first show in Listing 3 a *broken* implementation of a hub. Instead of flooding a packet out all ports except the input, this program incorrectly sends each packet received back out the *same* port on which it arrived. When applying the test scenario to this program, the Switchyard framework gives output similar to that shown in Figure 2. We see in the figure that Switchyard reports that one test passed (the first packet input event expectation), 1 failed (the first output expectation was not met), and that there are 6 tests that could not be evaluated due to a test failure. The output is normally color-coded to give a visual indication of test passage (green) and/or failure (red). When a test does not pass, Switchyard can optionally execute the Python debugger in such a way as to start the debugger as close to the part of the user program that failed as possible. This capability is enabled with a command-line switch.

We also see in the test output quite a bit of detail related to different expectation events and individual packets that were either received or sent. Switchyard can optionally show *full* packet dumps instead of abbreviated contents (which is the default) when tests pass or fail. Any explanatory text for a given test is created as part of a test scenario and supplied verbatim from the original scenario so the writer of a test can supply as much or as little context as he or she wishes, in order to help students to understand why a test may pass or fail.

When the broken hub program in Listing 3 is fixed to correctly flood a packet out all interfaces except the input, the output of Switchyard becomes green to reflect the fact that all the tests now pass. The partial test output for a correct hub implementation is shown in Figure 3. Once all tests pass successfully, the code can be refactored to improve its design and structure while preserving the correct functionality in a cycle commonly known as "red-green-refactor".

4. SWITCHYARD IN ACTION

In this section, we describe the set of assignments that we have used thus far with Switchyard, and discuss student experiences with using Switchyard in an introductory undergraduate networking course.

```
def srpy_main(net):
    while True:
        try:
            dev,packet = net.recv_packet()
        except Shutdown:
            return
        net.send_packet(dev, packet)
    net.shutdown()
```

Listing 3: Switchyard program for implementing a hub device (broken).

```
Results for test scenario hub tests: 1 passed, 1 failed,
6 pending

Passed:
1   An Ethernet frame with a broadcast destination address
    should arrive on eth1
        Expected event: recv_packet
        [30:00:00:00:00:02>ff:ff:ff:ff:ff:ff IP] on eth1

Failed:
    The Ethernet frame with a broadcast destination address
    should be forwarded out ports eth0 and eth2
        Expected event: send_packet(s)
        [30:00:00:00:00:02>ff:ff:ff:ff:ff:ff IP] out eth2 and
        [30:00:00:00:00:02>ff:ff:ff:ff:ff:ff IP] out eth0

Pending (couldn't test because of prior failure):
1   An Ethernet frame from 20:00:00:00:00:01 to
    30:00:00:00:00:02 should arrive on eth0
        Expected event: recv_packet
        [20:00:00:00:00:01>30:00:00:00:00:02 IP] on eth0
2   Ethernet frame destined for 30:00:00:00:00:02 should be
    flooded out eth1 and eth2
        Expected event: send_packet(s)
        [20:00:00:00:00:01>30:00:00:00:00:02 IP] out eth2 and
        [20:00:00:00:00:01>30:00:00:00:00:02 IP] out eth1
```

Figure 2: Code and output snippet for failed test

4.1 Assignments

We have used Switchyard as a basis for four projects of varying scope and difficulty in an undergraduate networking course, and have plans for additional projects and activities.

Learning Switch. Developing an Ethernet learning switch is quite simple using Switchyard (about 40 lines of code). This exercise was used as an introduction to Switchyard and its APIs, to executing and understanding test output, and to running Switchyard code on a Linux (virtual machine) host in Mininet. Students were supplied with a comprehensive set of tests for this project since they were unfamiliar with the APIs.

```
Results for test scenario hub tests: 8 passed, 0 failed,
0 pending

Passed:
1   An Ethernet frame with a broadcast destination address
    should arrive on eth1
        Expected event: recv_packet
        [30:00:00:00:00:02>ff:ff:ff:ff:ff:ff IP] on eth1
2   The Ethernet frame with a broadcast destination address
    should be forwarded out ports eth0 and eth2
        Expected event: send_packet(s)
        [30:00:00:00:00:02>ff:ff:ff:ff:ff:ff IP] out eth2 and
        [30:00:00:00:00:02>ff:ff:ff:ff:ff:ff IP] out eth0
```

Figure 3: Output snippet for two passed tests

IP Router. Students developed a full IPv4 router over three progressive stages. The first stage required students to develop code for responding to ARP requests, and also served as an introduction to the router projects. The second stage required students to generate ARP requests and handle ARP responses, and to implement IP routing (longest prefix match) using a statically configured forwarding table. The third stage required students to respond to ICMP echo requests ("pings") as well as generate ICMP errors such as time exceeded messages and network/host/port unreachable messages. Creating a fully functional IPv4 router involves quite a bit of technical complexity, and students were provided with several tests for each stage to help guide them through the development process. Students were also provided with scripts for running their router in a Mininet environment in order to observe its behavior (*e.g.*, using `traceroute` and other "real" tools).

Firewall. This project was designed as an add-on to the IPv4 router project, but could easily be designed differently. Students developed firewall functionality for their router based on a simple rule syntax. In addition to accept/deny firewall processing, students also implemented a token bucket rate limitation feature for rules that had specified rate limits. Students were given guidance for writing tests, but were not provided with any automated Switchyard tests.

Deep-packet Inspection Middlebox. For this project, students developed two network components: a Openflow-based switching element that would forward packets from pre-specified flows to a "deep-packet inspection" (DPI) device and forward other flows normally, and a DPI middlebox that inspected and optionally rewrote application-layer content if certain "suspicious" strings were found. When application content was modified, fields in lower-layer packet headers also needed to be amended (*i.e.*, length fields). This project gave students some exposure to SDN protocols and non-standard packet forwarding, and also provided the basis for discussing current news events related to government spying.

Additionally, we intend to create a new introductory-level exercise oriented around understanding packet headers by having students build a Wireshark-like tool, and we have plans to create several more projects including adding a dynamic routing stage to the IP router, a network/port address translator, and an IPv4–IPv6 gateway. Currently, all the exercises are oriented around networking functionality at intermediate devices such as switches, routers, and middleboxes, but we are planning to expand Switchyard to accommodate projects oriented around end-host functionality such as transport and application layer capabilities.

4.2 Classroom experiences

Thus far, qualitative and some quantitative feedback have been collected from students through surveys. Responses to these surveys have been overwhelmingly positive. Students reported that the system-building activities were very beneficial to their learning and developing a better understanding of networking concepts. Student evaluations also indicated very low-levels of frustration, which we attribute to the test-driven nature of development and the potential for students to focus efforts on passing one test at a time. Indeed, several students expressed that they wished they had been able to tackle additional system-building projects.

Two specific quantitative questions we asked were: (Q1) *To what degree did the router and switch projects help your understanding of course concepts* (e.g., *link-layer and network addressing, packet headers and encapsulation, IP forwarding, routing, etc.)?*, and (Q2) *To what degree did the test-driven nature of SRPY projects help with the projects?* For each of these questions, students responses were based on a 1–5 scale (1=not helpful, 3=moderately helpful, 5=extremely helpful). We received 23 responses to these

questions. For Q1, the student response average was 4.38 ($\sigma = 0.96$) and for Q2, the student response average was 4.69 ($\sigma = 0.48$).

Common themes that emerged from other (qualitative feedback) questions were that students felt that the router-building projects were "cool" and that the testing and debugging facilities in Switchyard were "essential" to them being able to accomplish the various tasks to build the router. One student noted that "the router projects were hard due to the inherent complexity, not because of [Switchyard] or [Python]", and another student captured the feeling of several students, that "the projects were not easy, and when we struggled it was mostly with conceptual topics". Similarly, another student wrote that "[Switchyard] was easy to learn and struggling with the concepts a little was actually helpful."We view the fact that students reported that when they struggled it was with networking concepts rather than the Switchyard API or language details as particularly encouraging.

Lastly, the responses suggested some areas for improving the APIs and pedagogical approach with Switchyard. First, although Python is taught in our CS1 course, a couple students did remark that they felt their biggest problem was Python syntax. Also, although the automated tests were generally viewed positively, one student noted that they felt like the tests made the projects feel too easy. There is certainly a balance to be struck between providing too few and too many tests, and having students write more or less tests. Most students felt that the projects still contained substantial difficulty, but the number of tests given to students may need to be modified semester-to-semester, depending on the group of students in a course. Finally, three students noted that the packet construction and parsing libraries were sometimes difficult to work with. The version of Switchyard used by the students leveraged the packet library from the POX Openflow controller, which does not prevent a variety of common error patterns. One particular problem is that new attributes (instance variables) can be added to objects dynamically. As a result, a misspelled attribute (*e.g.*, using `source` instead of `src` as the attribute for assigning an Ethernet source address) becomes a difficult-to-detect *logic* error. We completely revised the packet parsing and creation library in Switchyard to prevent many of the problems we observed students to have, using Python language features and implementation patterns. Among other improvements, misspelled attributes now result in an obvious runtime exception.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we describe a new Python-based framework called Switchyard for building networked systems such as switches, routers, and middleboxes. By virtue of using a higher-level language (Python) and through careful design of its APIs and libraries, Switchyard helps undergraduate students avoid pitfalls associated with low-level implementation languages, while still enabling them to build *real* systems. Switchyard supports test-driven development so that students can be given a set of tests or tasked to create them, and the framework seamlessly supports execution of student programs within the test harness or in a Linux environment. We describe our initial experiences with using Switchyard in an undergraduate networking course, and discuss how these experiences have shaped our ongoing efforts to improve the framework and libraries.

Our plans for future development of Switchyard include further simplifying and error-proofing its APIs and libraries and significantly expanding the set of exercises and materials available. We also plan to simplify execution of Switchyard in various environments (*e.g.*, by automating installation of `iptables`/`ebtables` rules as appropriate). Lastly, we plan to improve forwarding performance so that Switchyard can effectively be used in Gigabit-speed environments.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grant 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.

6. REFERENCES

[1] Openflow switch specification, version 1.0.0. `http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf`, 2009.

[2] About POX. `http://www.noxrepo.org/pox/about-pox/`, 2014.

[3] Network Simulator ns-3. `http://www.nsnam.org`, 2014.

[4] Wisconsin Advanced Internet Laboratory. `http://wail.cs.wisc.edu`, 2014.

[5] M. Casado and N. McKeown. The virtual network system. *ACM SIGCSE Bulletin*, 37(1), 2005.

[6] D. Yuan *et al.* An instructional design of open source networking laboratory and curriculum. In *Proceedings of the 10th ACM SIGITE*, 2009.

[7] N. Feamster and J. Rexford. Getting students' hands dirty with clean-slate networking. In *ACM SIGCOMM Education Workshop*, 2011.

[8] M. Goldweber and R. Davoli. VDE: an emulation environment for supporting computer networking courses. *ACM SIGCSE Bulletin*, 40(3), 2008.

[9] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of CoNEXT '12*, 2012.

[10] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale Virtualization in the Emulab Network Testbed. In *USENIX Annual Technical Conference*, 2008.

[11] W. D. Laverell, Z. Fei, and J. N. Griffioen. Isn't it time you had an Emulab? *SIGCSE Bull.*, 40, March 2008.

[12] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA—An Open Platform for Gigabit-Rate Network Switching and Routing. In *IEEE International Conference on Microelectronic Systems Education*, pages 160–161, 2007.

[13] M. Erlinger *et al.* Tinkernet: a low-cost networking laboratory. In *Proceedings of the sixth ACE*, 2004.

[14] M. Maheswaran, *et al.* GINI: a user-level toolkit for creating micro internets for teaching & learning computer networking. In *Proceedings of the 40th ACM SIGCSE*, 2009.

[15] J. Pan. Teaching computer networks in a real network: the technical perspectives. In *Proceedings of the 41st ACM SIGCSE*. ACM, 2010.

[16] M. Pizzonia and M. Rimondini. Netkit: easy emulation of complex networks on inexpensive hardware. In *Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*, 2008.

[17] S. Yoo *et al.* Remote access internetworking laboratory. In *Proceedings of the 35th ACM SIGCSE*, 2004.

[18] S. Savage. Sting: A TCP-based Network Measurement Tool. In *USENIX Symposium on Internet Technologies and Systems*, volume 2, 1999.

[19] J. Sommers, P. Barford, and W. Willinger. A Proposed Framework for Calibration of Available Bandwidth Estimation Tools. In *Proceedings of IEEE Symposium on Computers and Communication*, June 2006.

[20] C. Wiseman, K. Wong, T. Wolf, and S. Gorinsky. Operational experience with a virtual networking laboratory. *SIGCSE Bull.*, 40, March 2008.

[21] K. Wong, T. Wolf, S. Gorinsky, and J. Turner. Teaching experiences with a virtual network laboratory. *SIGCSE Bull.*, 39, March 2007.