Switchyard Documentation

Release 2017.01.4

Joel Sommers

Jan 12, 2017

CONTENTS

CHAPTER

ONE

INTRODUCTION AND OVERVIEW

Switchyard is a framework for creating, testing, and experimenting with software implementations of networked systems such as Ethernet switches, IP routers, firewalls and middleboxes, and end-host protocol stacks. Switchyard can be used for system-building projects targeting layers of the network protocol stack from layer 2 (link layer) and above. It is intended primarily for educational use and has purpose-built testing and debugging features. Although its design favors understandability over speed, it can work quite nicely as a prototyping environment for new kinds of networked devices.

The Switchyard framework is implemented in Python and consists of two components: a program (`swyard`) which creates a runtime environment for the code that implements some networked system or device, and a collection of library modules that can be used for a variety of tasks such as packet creation and parsing. The networked system code is implemented in one or more Python files (which you write!) and that use the Switchyard libraries and conform to certain conventions. The `swyard` runtime environment creator and orchestrator seamlessly handles running your code either in a test setting where no actual network traffic is generated or in a real or "live" setting in which your code can interact with other networked systems.

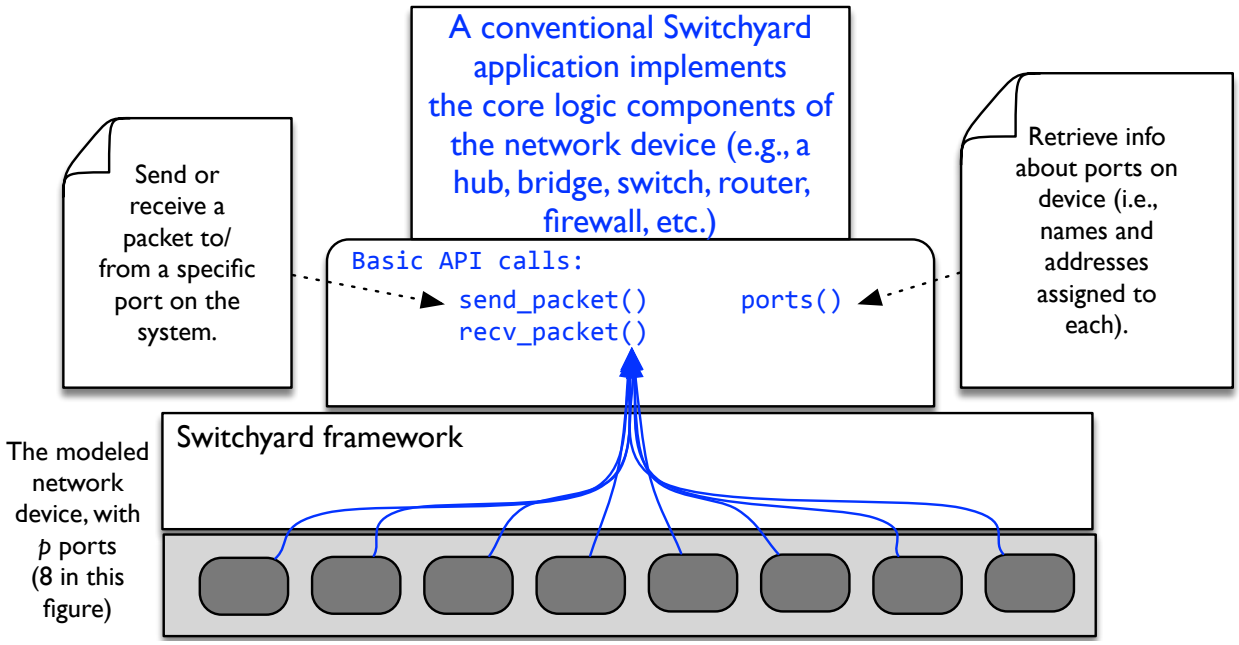
The Switchyard runtime environment (depicted below) provides a given networked system with 1 or more *interfaces* or *ports*. A port may represent a wired connection to another device, or may represent a wireless interface, or may represent a *loopback*[1] interface. In any case, it is through these ports that packets are sent and received. Each port has, at minimum, a name (e.g., `en0`) and an Ethernet address. A port may also have an IPv4 address and subnet mask associated with it.

The typical goal of a Switchyard-based program is to receive a packet on one port, possibly modify it, then either forward it out one or more ports or to drop the packet. The rest of this documentation is organized around how to perform these tasks in various settings. In particular:

- The next section, *writing a Switchyard program*, describes how to develop a basic Switchyard program, including what APIs are available for parsing and constructing packets and sending/receiving packets on network interfaces.
- Following that, the next section, *running in the test environment*, provides details on running a Switchyard program in the test environment. The section after that gives details on *how to write test scenarios*.
- The next section describes *running Switchyard in a live environment*, such as on a standard Linux host or within the Mininet emulation environment or some other kind of virtual environment.
- *Advanced API topics* are addressed next, such as creating new packet header types, and implementing network protocol stacks that can interoperate with a Python socket-based program.
- An *installation guide* appears next.
- Finally, you can find an *API Reference* at the end of this documentation along with and an index.

A note to the pedantic: In this documentation we use the term *packet* in a generic sense to refer to what may more traditionally be a link layer *frame*, a network layer *packet*, a transport layer *segment*, or an application

[1] The loopback interface is a *virtual* interface that connects a host to itself. It is typically used to facilitate network communication among processes on the same host.



layer *message*. Where appropriate, we use the appropriate specific term, but often resort to using *packet* in a more general sense.

And one more (genuinely important) note: Switchyard is Python 3-only! You'll get an error (or maybe even more than one error!) if you try to use Switchyard with Python 2. Python 3.4 is required, at minimum. An installation guide (see *Installing Switchyard*) is provided in this documentation to help with getting any necessary libraries installed on your platform to make Switchyard work right.

CHAPTER

TWO

WRITING A SWITCHYARD PROGRAM

A Switchyard program is simply a Python program that includes a particular entrypoint function which accepts a single parameter. The startup function can simply be named `main`, but can also be named `switchy_main` if you like. The function must accept at least one parameter, which is a reference to the Switchyard *network object* (described below). Method calls on the network object are used to send and receive packets to and from network ports.

A Switchyard program isn't executed *directly* with the Python interpreter. Instead, the program `swyard` is used to start up the Switchyard framework and to load your code. When Switchyard starts your code it looks for a function named `main` and invokes it, passing in the network object as the first parameter. Details on how to start Switchyard (and thus your program) are given in the chapters on *running a Switchyard in the test environment* and *running Switchyard in a live environment*. Note that it is possible to pass arguments into a Switchyard program; see *Passing arguments into a Switchyard program* for details.

A Switchyard program will typically also import other Switchyard modules such as modules for parsing and constructing packets, dealing with network addresses, and other functions. These modules are introduced below and described in detail in the *API reference chapter*.

2.1 Introducing the "network object"

As mentioned above, a Switchyard program can simply have a `main` function that accepts a single argument. The parameter passed to `main` is called the "network object". It is on this object that you can call methods for sending and receiving packets and getting information about ports on the device for which you're implementing the logic.

2.1.1 Sending and receiving packets

As a way to describe two of the most important methods on the network object, here is a program that receives one packet, prints it out, sends it *back out the same interface*, then quits.

Notice in the code below that we only need to import *`switchyard.lib.userlib`* to get access to various Switchyard classes and functions; generally speaking, this is the *only* import you should ever need for any Switchyard program. Although you can import individual Switchyard modules separately (for the specific module to import, see *API Reference*), but you will probably find that importing `userlib` is much easier.

```
from switchyard.lib.userlib import *

def main(net):
    timestamp,input_port,packet = net.recv_packet()
    print ("Received {} on {}".format(packet, input_port))
    net.send_packet(input_port, packet)
```

This program isn't likely to be very useful — it is just meant as an illustration of the most important two methods on the network object:

- `recv_packet(timeout=None)`

  Not surprisingly, this method is used to receive at most one packet from any port. The method will *block* until a packet is received, unless a timeout value >=0 is given. The default is to block indefinitely. The method returns a *namedtuple* of length 3, which includes a timestamp for when the packet was received, the name of the input port on which the packet was received, and the packet itself (another example is given below, plus see `collections.namedtuple` in the Python library reference).

  The method raises a `Shutdown` exception if the Switchyard framework has been shut down. It can also raise a `NoPackets` exception if no packets are received before the timeout value given to the method expires.

- `send_packet(output_port, packet)`

  Again, the meaning of this method call is probably not especially surprising: when called, the given packet will be sent out the given output port. For the `output_port` parameter, the string name of the port can be given, or an `Interface` object may also be supplied (see below for *more about Interface objects* as well as the *Interface and InterfaceType reference*).

  This method returns `None`. If the `output_port` or some detail about the given packet is invalid (e.g., something other than a packet is passed as the second parameter), this method raises a `ValueError`.

Returning briefly to the `recv_packet` method, observe that in the above example no arguments are given so the call will block until a packet is received. Also, it is important to recognize that the return type of `recv_packet` is a *namedtuple* of exactly three elements so in addition to automatically unpacking the tuple as in the above example, you can use indexing or attribute-like syntax on the return value from `recv_packet`. For example (using attribute-syntax):

```
from switchyard.lib.userlib import *

def main(net):
    # below, recvdata is a namedtuple
    recvdata = net.recv_packet()
    print ("At {}, received {} on {}".format(
        recvdata.timestamp, recvdata.packet, recvdata.input_port))

    # alternatively, the above line could use indexing, although
    # readability suffers:
    #    recvdata[0], recvdata[2], recvdata[1]))

    net.send_packet(recvdata.input_port, recvdata.packet)

    # likewise, the above line could be written using indexing
    # but, again, readability suffers:
    # net.send_packet(recvdata[1], recvdata[2])
```

Importantly, note that in the above examples we are not handling any potential exceptions that could occur. In particular, we really should be handling *at least* the situation in which the framework is shut down (and we receive a `Shutdown` exception). Just for completeness, we should also handle the `NoPackets` exception, although if the code is designed to block indefinitely we shouldn't normally receive that particular exception.

Let's rewrite the code above, and now put everything in a `while` loop so that we keep reading and sending packets as long as we're running. We will eventually turn this code into a working network *hub* implementation[1], but it's currently broken because it still just sends a packet out the *same port* on which it

[1] A hub is a network device with multiple physical ports. Any packet to arrive on a port is sent back out *all* ports **except** for the one on which it arrived.

arrived:

```
from switchyard.lib.userlib import *

def main(net):
    while True:
        try:
            timestamp,input_port,packet = net.recv_packet()
        except Shutdown:
            log_info ("Got shutdown signal; exiting")
            break
        except NoPackets:
            log_info ("No packets were available.")
            continue

        # if we get here, we must have received a packet
        log_info ("Received {} on {}".format(packet, input_port))
        net.send_packet(input_port, packet)
```

In the example above, notice that we also changed the `print` function calls to `log_info`. Switchyard uses built-in Python logging capabilities (see `logging` in the Python library reference) for printing various notices to the console. The *logging functions*, described below, each just accept one string parameter which is just the text to be printed on the console.

For full details of the `send_packet` and `recv_packet` method calls, refer to *Net object reference* in the *API Reference* section at the end of this documentation.

2.1.2 Getting information about ports (interfaces) on the device

Other methods available the network object relate to getting information about the ports/interfaces attached to the device on which the Switchyard code is running. The two basic methods are `interfaces` and `ports`. These methods are aliases and do exactly the same thing. In particular:

- `interfaces()`

  This method returns a list of interfaces that are configured on the network device, as a list of `Interface` objects. The alias method `ports()` does exactly the same thing. There is no inherent ordering to the list of `Interface` objects returned.

Each `Interface` object has a set of properties that can be used to access various configured attributes for the interface:

- `name`: returns the name of the interface (e.g., `en0`) as a string.
- `ethaddr`: returns the Ethernet address associated with the interface, as a *`switchyard.lib.address.EthAddr`* instance.
- `ipaddr`: returns the IPv4 address associated with the interface, if any. This property returns an object of type `IPv4Address`. If there is no address assigned to the interface, the address is 0.0.0.0. A current limitation with the `Interface` implementation in Switchyard is that only one address can be associated with an interface, and it must be an IPv4 address. Eventually, Switchyard will fully support IPv6 addresses, and multiple IP addresses per interface.
- `netmask`: returns the network mask associated with the IPv4 address assigned to the interface. The netmask defaults to 255.255.255.255 (/32) if none is specified.
- `ifnum`: returns an integer index associated with the interface.
- `iftype`: returns the type of the interface, if it can be inferred by Switchyard. The return type is a value from the *`switchyard.lib.interface.InterfaceType`* enumerated type. The type can either be

`Unknown`, `Loopback`, `Wired`, or `Wireless`. The type is automatically set when an interface is initialized. Note that in some cases the type can be inferred, but in others it cannot (thus the potential for an `Unknown` value).

All the above properties except `ifnum` and `iftype` are modifiable. Changing them can be accomplished just by assigning a new value to the property. Beware, though, that changing address values has no effect on the underlying host operating system if Switchyard is run in a live environment, so you would generally be wise to leave the addresses alone.

For full interface details, see *Interface and InterfaceType reference*.

As an example, to simply print out information regarding each interface defined on the current network device you could use the following program:

```
def main(net):
    for intf in net.interfaces():
        log_info("{} has ethaddr {} and ipaddr {}/{} and is of type {}".format(
            intf.name, intf.ethaddr, intf.ipaddr, intf.netmask, intf.iftype.name))

    # could also be:
    # for intf in net.ports():
    #    ...
```

Entirely depending on how the network device is configured, output from the above program might look like the following:

```
09:10:08 2016/12/17     INFO eth0 has ethaddr 10:00:00:00:00:01 and ipaddr 172.16.42.1/255.255.255.
↪252 and is of type Unknown
09:10:08 2016/12/17     INFO eth1 has ethaddr 10:00:00:00:00:02 and ipaddr 10.10.0.1/255.255.0.0␣
↪and is of type Unknown
09:10:08 2016/12/17     INFO eth2 has ethaddr 10:00:00:00:00:03 and ipaddr 192.168.1.1/255.255.255.
↪0 and is of type Unknown
```

The above example code was run in the *Switchyard *test* environment*; when a Switchyard program is run in test mode, all interfaces will show type `Unknown`. Note also that there is *no inherent ordering* to the list of interfaces returned.

There are a few convenience methods related to `ports` and `interfaces`, which can be used to look up a particular interface given a name, IPv4 address, or Ethernet (MAC) address:

- `interface_by_name(name)`: This method returns an `Interface` object given a string name of a interface. An alias method `port_by_name(name)` also exists.
- `interface_by_ipaddr(ipaddr)`: This method returns an `Interface` object given an IP address configured on one of the interfaces. The IP address may be given as a string or as an IPv4Address object. An alias method `port_by_ipaddr(ipaddr)` also exists.
- `interface_by_macaddr(ethaddr)`: This method returns an `Interface` object given an Ethernet (MAC) address configured on one of the interfaces. An alias method `port_by_macaddr(ethaddr)` also exists.

Note that the above lookup methods raise a `KeyError` exception if the lookup name is invalid.

2.1.3 Other methods on the network object

Lastly, there is a `shutdown` method available on the network object. This method should be used by a Switchyard program prior to exiting in order to clean up and shut down various resources.

Let's now add a bit to the previous example program to turn it into an almost-complete implementation of a hub. Whenever we receive a packet, we need to loop through the ports on the device and send the packet

on a port as long as the port isn't the one on which we received the packet (lines 21-23, below):

Listing 2.1: A (nearly) full implementation of a hub.

```
1  from switchyard.lib.userlib import *
2
3  def main(net):
4      # add some informational text about ports on this device
5      log_info ("Hub is starting up with these ports:")
6      for port in net.ports():
7          log_info ("{}: ethernet address {}".format(port.name, port.ethaddr))
8
9      while True:
10         try:
11             timestamp,input_port,packet = net.recv_packet()
12         except Shutdown:
13             # got shutdown signal
14             break
15         except NoPackets:
16             # try again...
17             continue
18
19         # send the packet out all ports *except*
20         # the one on which it arrived
21         for port in net.ports():
22             if port.name != input_port:
23                 net.send_packet(port.name, packet)
24
25     # shutdown is the last thing we should do
26     net.shutdown()
```

There's still one thing missing from the above code, which is for the hub to ignore any frames that are destined to the hub itself. That is, if an Ethernet destination address in a received frame is the same as an Ethernet address assigned to one of the ports on the hub, the frame should *not* be forwarded (it can simply be ignored). Finishing off the hub by doing this is left as an exercise.

2.2 Introduction to packet parsing and construction

This section provides an overview of packet construction and parsing in Switchyard. For full details on these capabilities, see *Packet parsing and construction reference*.

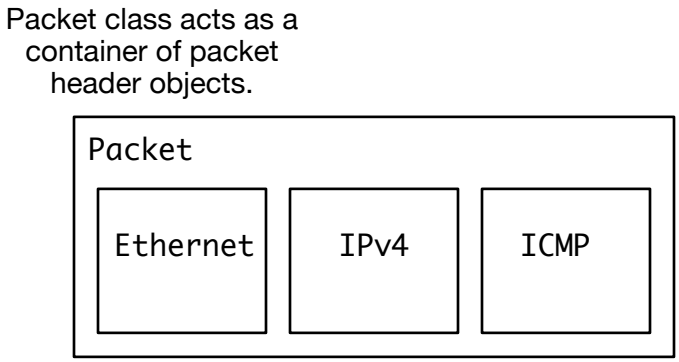
Switchyard's packet construction/parsing library is found in `switchyard.lib.packet`. Its design is based on a few other libraries out there, including POX's library[2] and Ryu's library[3].

There are a few key ideas to understand when using the packet library:

- The `Packet` class acts as a container of headers (or rather, of header objects).
- Headers within a packet can be accessed through methods on the Packet container object, and also by indexing. Headers are ordered starting with lowest layer protocols. For example, if a `Packet` has an `Ethernet` header (which is likely to be the lowest layer protocol), this header can be accessed with index 0 as in `pktobj[0]`. Indexes can be integers, and they can also be packet header class names (e.g., `Ethernet`, `IPv4`, etc.). For example, to access the `Ethernet` header of a packet, you can write `pktobj[Ethernet]`.

[2] https://github.com/noxrepo/pox
[3] https://github.com/osrg/ryu

- Fields in header objects are accessed through standard Python *properties*. The code to manipulate header fields thus looks like it is just accessing instance variables, but "getter" and "setter" method calls actually take place, depending on whether a property is being retrieved or assigned to.
- A packet object can be constructed by either expliciting instantiating an object and adding headers, or it can be formed by "adding" (using the + operator) headers together, or by appending headers onto a packet (using + or +=).
- The Switchyard framework generally *automatically* handles serializing and deserializing Packet objects to and from byte sequences (i.e., wire format packets), but you can also explicitly invoke those methods if you need to.



Here are some examples using `Ethernet`, `IPv4`, and `ICMP` headers. First, let's construct a packet object and add these headers to the packet:

```
>>> from switchyard.lib.packet import *
>>> p = Packet()    # construct a packet object
>>> e = Ethernet() # construct Ethernet header
>>> ip = IPv4()    # construct IPv4 header
>>> icmp = ICMP()  # construct ICMP header
>>> p += e         # add eth header to packet
>>> p += ip        # add ip header to packet
>>> p += icmp      # add icmp header to packet
>>> print (p)
Ethernet 00:00:00:00:00:00->00:00:00:00:00:00 IP | IPv4 0.0.0.0->0.0.0.0 ICMP | ICMP EchoRequest 0␣
↪0 (0 data bytes)
```

A shorthand for doing the above is:

```
>>> p = Ethernet() + IPv4() + ICMP()
```

The effect of the + operator with header objects as in the previous line is to construct a packet object, just as the first example. Note that with the above one-line example, the default Ethertype for the Ethernet header is IPv4, and the default protocol number for IPv4 is ICMP. Thus, this example is somewhat special in that we didn't need to modify any of the packet header fields to create a (mostly) valid packet. Lastly, note that the order in which we add packet headers together to construct a full packet is important: lower layers (e.g., `Ethernet`) must come first, followed by other protocol headers in their correct order.

Switchyard does *not* ensure that a constructed Packet is sensible in any way. It is possible to put headers in the wrong order, to supply illogical values for header elements (e.g., a protocol number in the IPv4 header that doesn't match the next header in the packet), and to do other invalid things. Switchyard gives you the tools for constructing packets, but doesn't tell you how to do so.

The `num_headers` Packet method returns the number of headers in a packet, which returns the expected number for this example:

```
>>> p.num_headers()
3
```

Note that the `len` function on a packet returns the number of *bytes* that the Packet would consume if it was in wire (serialized) format. The `size` method returns the same value.

```
>>> len(p)
42
>>> p.size()
42
```

(Note: Ethernet header is 14 bytes + 20 bytes IP + 8 bytes ICMP = 42 bytes.)

Packet header objects can be accessed conveniently by indexing. Standard negative indexing also works. For example, to obtain a reference to the Ethernet header object and to inspect and modify the Ethernet header, we might do the following:

```
>>> p[0] # access by index
<switchyard.lib.packet.ethernet.Ethernet object at 0x104474248>
>>> p[0].src
EthAddr('00:00:00:00:00:00')
>>> p[0].dst
EthAddr('00:00:00:00:00:00')
>>> p[0].dst = "ab:cd:ef:00:11:22"
>>> str(p[0])
'Ethernet 00:00:00:00:00:00->ab:cd:ef:00:11:22 IP'
>>> p[0].dst = EthAddr("00:11:22:33:44:55")
>>> str(p[0])
'Ethernet 00:00:00:00:00:00->00:11:22:33:44:55 IP'
>>> p[0].ethertype
<EtherType.IP: 2048>
>>> p[0].ethertype = EtherType.ARP
>>> print (p)
Ethernet 00:00:00:00:00:00->00:00:00:00:00:00 ARP | IPv4 0.0.0.0->0.0.0.0 ICMP | ICMP EchoRequest␣
↪0 0 (0 data bytes)
>> p[0].ethertype = EtherType.IPv4 # set it back to sensible value
```

Note that all header field elements are accessed through *properties*. For Ethernet headers, there are three properties that can be inspected and modified, `src`, `dst` and `ethertype`, as shown above. Notice also that Switchyard doesn't prevent a user from setting header fields to illogical values, e.g., when we set the ethertype to ARP although the next header is IPv4, not ARP. All `EtherType` values are specified in `switchyard.lib.packet.common`, and imported when the module `switchyard.lib.packet` is imported.

Accessing header fields in other headers works similarly. Here are examples involving the IPv4 header:

```
>>> p.has_header(IPv4)
True
>>> p.get_header_index(IPv4)
1
>>> str(p[1]) # access by index
'IPv4 0.0.0.0->0.0.0.0 ICMP'
>>> str(p[IPv4]) # access by header type
'IPv4 0.0.0.0->0.0.0.0 ICMP'
>>> p[IPv4].protocol
<IPProtocol.ICMP: 1>
>>> p[IPv4].src
```

```
IPv4Address('0.0.0.0')
>>> p[IPv4].dst
IPv4Address('0.0.0.0')
>>> p[IPv4].dst = '149.43.80.13'
```

IPv4 protocol values are specified in `switchyard.lib.packet.common`, just as with `EtherType` values. Note, however, that you do not need to explicitly import this module if you import `switchyard.lib.userlib` — packet-related classes and enumerations are imported when importing `userlib`. The full set of properties that can be manipulated in the IPv4 header as well as all other headers is described in the *reference documentation for the packet library*.

Lastly, an example with the ICMP header shows some perhaps now familiar patterns. The main difference with ICMP is that the "data" portion of an ICMP packet changes, depending on the ICMP type. For example, if the type is 8 (ICMP echo request) the ICMP data becomes an object that allows the identifier and sequence values to be inspected and modified.

```
>>> p.has_header(ICMP)
True
>>> p.get_header_index(ICMP)
2
>>> p[2] # access by index; notice no conversion to string
<switchyard.lib.packet.icmp.ICMP object at 0x104449c78>
>>> p[ICMP] # access by header type
<switchyard.lib.packet.icmp.ICMP object at 0x104449c78>
>>> p[ICMP].icmptype
<ICMPType.EchoRequest: 8>
>>> p[ICMP].icmpcode
<EchoRequest.EchoRequest: 0>
>>> p[ICMP].icmpdata
<switchyard.lib.packet.icmp.ICMPEchoRequest object at 0x1044742c8>
>>> icmp.icmpdata.sequence
0
>>> icmp.icmpdata.identifier
0
>>> icmp.icmpdata.identifier = 42
>>> icmp.icmpdata.sequence = 13
>>> print (p)
Ethernet 00:00:00:00:00:00->00:11:22:33:44:55 IP | IPv4 0.0.0.0->149.43.80.13 ICMP | ICMP␣
↪EchoRequest 42 13 (0 data bytes)
```

By default, no "payload" data are included in with an ICMP header, but we can change that using the `data` property on the icmpdata part of the header:

```
>>> icmp.icmpdata.data = "hello, world"
>>> print (p)
Ethernet 00:00:00:00:00:00->00:11:22:33:44:55 IP | IPv4 0.0.0.0->149.43.80.13 ICMP | ICMP␣
↪EchoRequest 42 13 (12 data bytes)
```

Python keyword argument syntax can be used to assign values to header fields when a header object is constructed. This kind of syntax can make packet construction a bit more compact and streamlined. For example, if we wanted to make a UDP packet with some payload, we could do something like the following:

```
>>> e = Ethernet(src="11:22:33:44:55:66", dst="66:55:44:33:22:11", ethertype=EtherType.IP)
>>> ip = IPv4(src="1.2.3.4", dst="4.3.2.1", protocol=IPProtocol.UDP, ttl=32)
>>> udp = UDP(src=1234, dst=4321)
>>> p = e + ip + udp + b"this is some application payload!"
>>> print(p)
```

```
Ethernet 11:22:33:44:55:66->66:55:44:33:22:11 IP | IPv4 1.2.3.4->4.3.2.1 UDP | UDP 1234->4321 |␣
↪RawPacketContents (33 bytes) b'this is so'...
>>>
```

Finally, to serialize the packet into a wire format sequence of bytes, we can use the `to_bytes()` method:

```
>>> p.to_bytes()
b'\x00\x11
↪"3DU\x00\x00\x00\x00\x00\x08\x00E\x00\x00(\x00\x00\x00\x00\x00\x01\xba\xd6\x00\x00\x00\x00\x00\x00\x00\x00\x0
↪ world'
```

Switchyard normally handles *deserialization* automatically, but there is a `from_bytes(raw)` method available that accepts a raw bytes object and reconstructs packet header attributes. It either succeeds or throws an exception. It returns any bytes that were not necessary for reconstructing the header.

As mentioned above, Switchyard does not require packets to be correctly constructed (e.g., there may be a TCP header in a packet without any IP header). As a result, while serialization will often succeed even if the packet is malformed, whereas deserialization often will not. The reason is that in deserialization, the contents of bytes earlier in a packet are necessary for determining how to reconstruct later headers and attributes in a packet (e.g., the `ethertype` attribute in the Ethernet header is necessary for determining which header comes next).

Other header classes that are available in Switchyard include `Arp`, `UDP`, `TCP`, `IPv6`, and `ICMPv6`. Again, see the *packet library reference documentation* for details on these header classes, and full documentation for all classes.

2.3 Utility functions

There are a few additional utility functions that are useful when developing a Switchyard program related to logging and debugging.

2.3.1 Logging functions

Switchyard uses Python's standard logging facilities and provides four convenience functions. Each of these functions takes a string as a parameter and prints it to the console as a logging message. The only difference with the functions relates to the logging *level* (see `logging` in the Python library reference), and whether the output is colored to visually highlight a problem. The default logging level is INFO within Switchyard. If you wish to include debugging messages, you can use the `-d` flag for the various invocation programs (e.g., `swyard`), as described in *Running in the test environment* and *Running in a "live" environment*.

log_debug(*str*)
: Write a debugging message to the log using the log level DEBUG.

log_info(*str*)
: Write a debugging message to the log using the log level INFO.

log_warn(*str*)
: Write a debugging message to the log using the log level WARNING. Output is colored magenta.

log_failure(*str*)
: Write a debugging message to the log using the log level CRITICAL. Output is colored red.

While you can still use the built-in `print` function to write messages to the console, using the log functions provides a much more structured way of writing information to the screen.

2.3.2 Invoking the debugger

Although a longer discussion of debugging is included in *a later section*, it is worth mentioning that there is a built-in function named `debugger` that can be used *anywhere* in Switchyard code to immediately invoke the standard Python pdb debugger.

For example, if we add a call to `debugger()` in the example code above just *after* the try/except block, then *run the code in a test environment*, the program pauses immediately after the call to debugger and the pdb prompt is shown:

```
# after hub code is started in test environment,
# some output is shown, followed by this:

> /Users/jsommers/Dropbox/src/switchyard/xhub.py(29)main()
-> for port in net.ports():
(Pdb) list
 24
 25                 debugger()
 26
 27                 # send the packet out all ports *except*
 28                 # the one on which it arrived
 29  ->             for port in net.ports():
 30                     if port.name != input_port:
 31                         net.send_packet(port.name, packet)
 32
```

As you can see, the program is paused on the next executable line following the call to `debugger()`. At this point, any valid `pdb` commands can be given to inspect or alter program state. Once again, see later sections for details on running Switchyard code *in a live environment* and on other *debugging capabilities*.

2.4 Passing arguments into a Switchyard program

It is possible to pass in additional arguments to a Switchyard program via its `main` function. To accept additional arguments into your `main` function, you should *at least* add a `*args` parameter. You can optionally also accept keyword-style arguments by including a `**kwargs` parameter. For example, here is the initial part of a `main` function which accepts both:

```
def main(netobj, *args, **kwargs):
    # args is a list of arguments
    # kwargs is a dictionary of key-value keyword arguments
```

As noted in the code comment, the parameter `*args` will collect any *non-keyword* arguments into a list and the parameter `**kwargs` will collect any keyword-style arguments into a dictionary. Note that *all* argument values are passed in as strings, so your program may need to do some type conversion.

To pass arguments into your `main` function from invoking `swyard` on the command line, use the `-g` option. This option accepts a string, which should include all arguments to be passed to your `main` function, each separated by spaces. For keyword-style arguments, you can use the syntax `param=value`. Any space-separated strings that do not include the = character as passed into the arglist (`args`). For example, to pass in the value `13` and the keyword parameter `debug=True`, you could use the following command-line:

```
$ swyard -g "13 debug=True" ... (other arguments to swyard)
```

When invoking your `main` function, `args` would have a single value (the string `'13'`) and `kwargs` would be the dictionary `{'debug': 'True'}` (notice that `True` would be a string since all arguments end up being passed in as strings).

CHAPTER

THREE

RUNNING IN THE TEST ENVIRONMENT

To run Switchyard in test mode, a *test scenario* file is needed. This file includes specifications of various events (sending particularly crafted packets, receiving packets, etc.) that a Switchyard program is expected to do if it behaves correctly. Also needed, of course, is the Switchyard program you wish to test. The test scenario files may be regular Python (`.py`) files, but they may alternatively have an extension `.srpy` if they have been *compiled*. For details on creating and compiling test scenarios, see *Test scenario creation*.

Let's say your program is named `myhub.py`. To invoke Switchyard in test mode and subject your program to a set of tests, at minimum you would invoke `swyard` as follows:

```
$ swyard -t hubtests.srpy myhub
```

Note that the `-t` option puts `swyard` in test mode. The argument to the `-t` option should be the name of the test scenario to be executed, and the final argument is the name of your code. It doesn't matter whether you include the `.py` extension on the end of your program name, so:

```
$ swyard -t hubtests.srpy myhub.py
```

would work the same as above.

3.1 Test output

When you run `swyard` in test mode and all tests pass, you'll see something similar to the following:

Listing 3.1: Abbreviated (normal) test output.

```
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
2   The Ethernet frame with a broadcast destination address
    should be forwarded out ports eth0 and eth2
3   An Ethernet frame from 20:00:00:00:00:01 to
    30:00:00:00:00:02 should arrive on eth0
4   Ethernet frame destined for 30:00:00:00:00:02 should be
    flooded out eth1 and eth2
5   An Ethernet frame from 30:00:00:00:00:02 to
    20:00:00:00:00:01 should arrive on eth1
6   Ethernet frame destined to 20:00:00:00:00:01 should be
    flooded out eth0 and eth2
7   An Ethernet frame should arrive on eth2 with destination
    address the same as eth2's MAC address
```

```
8   The hub should not do anything in response to a frame
    arriving with a destination address referring to the hub
    itself.

All tests passed!
```

Note that the above output is an abbreviated version of test output and is normally shown in colored text when run in a capable terminal.

3.2 Verbose test output

If you invoke `swyard` with the `-v` (verbose) option, the test output includes quite a bit more detail:

Listing 3.2: Verbose test output.

```
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 Ethernet
        30:00:00:00:00:02->ff:ff:ff:ff:ff:ff IP | IPv4
        172.16.42.2->255.255.255.255 ICMP | ICMP EchoRequest 0 0 (0
        data bytes) on eth1
2   The Ethernet frame with a broadcast destination address
    should be forwarded out ports eth0 and eth2
        Expected event: send_packet(s) Ethernet
        30:00:00:00:00:02->ff:ff:ff:ff:ff:ff IP | IPv4
        172.16.42.2->255.255.255.255 ICMP | ICMP EchoRequest 0 0 (0
        data bytes) out eth0 and Ethernet
        30:00:00:00:00:02->ff:ff:ff:ff:ff:ff IP | IPv4
        172.16.42.2->255.255.255.255 ICMP | ICMP EchoRequest 0 0 (0
        data bytes) out eth2
3   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 Ethernet
        20:00:00:00:00:01->30:00:00:00:00:02 IP | IPv4
        192.168.1.100->172.16.42.2 ICMP | ICMP EchoRequest 0 0 (0
        data bytes) on eth0

...
```

Note that the above output has been truncated — output would normally be shown for all tests. When invoked with the *verbose* option, individual tests show exactly what packets would be expected (either as input to a device or as output from it).

Test scenario descriptions that drive test executions as shown here are composed of a series of test *expectations*. Test expectations may be that a packet is received on a particular port, or that a packet is emitted out one or more ports, or that the user code calls `recv_packet` but times out (and thus nothing is received). Both the abbreviated and verbose test output shown above contain brief descriptions of the nature of each test. In the verbose output, packet details related to each test are also shown. Reading this information can help to understand what the tests are trying to accomplish, especially when a test expectation fails.

3.3 When a test fails

If some test expectation is not met, then the output indicates that something has gone wrong and, by default, Switchyard gives the user the standard Python pdb debugger prompt. The motivation for immediately putting the user in pdb is to enable just-in-time debugging. If the test output is read carefully and can be used to identify a flaw by inspecting code and data at the time of failure, then this should help to facilitate the development/testing/debugging cycle. At least that's the hope.

Say that we've done something wrong in our code which causes a test expectation to fail. The output we see might be similar to the following (note that to create the output below, we've used the full set of hub device tests, but the code we've used is the broken code we started with in *Writing a Switchyard program* that sends any packet back out the same port that it arrived on):

Listing 3.3: Normal (abbreviated) test output when one test fails.

```
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


Failed:
    The Ethernet frame with a broadcast destination address
    should be forwarded out ports eth0 and eth2
        Expected event: send_packet(s) Ethernet
        30:00:00:00:00:02->ff:ff:ff:ff:ff:ff IP | IPv4 | ICMP out
        eth0 and Ethernet 30:00:00:00:00:02->ff:ff:ff:ff:ff:ff IP |
        IPv4 | ICMP out eth2


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
2   Ethernet frame destined for 30:00:00:00:00:02 should be
    flooded out eth1 and eth2
3   An Ethernet frame from 30:00:00:00:00:02 to
    20:00:00:00:00:01 should arrive on eth1
4   Ethernet frame destined to 20:00:00:00:00:01 should be
    flooded out eth0 and eth2
5   An Ethernet frame should arrive on eth2 with destination
    address the same as eth2's MAC address
6   The hub should not do anything in response to a frame
    arriving with a destination address referring to the hub
    itself.

... (output continues)
```

Notice in the first line of output that Switchyard shows how many tests pass, how many have failed, and how many are *pending*. The pending category simply means that tests cannot be run because some earlier test failed. In the example above, the output from `swyard` clearly shows which test fails; when that happens, some additional explanatory text is shown, and a debugger session is started as close as possible to the point of failure. When not run in verbose mode, Switchyard will show abbreviated test descriptions for any passed tests and any pending tests, but the failed test will show everything.

Following the overall test results showing passed, failed, and pending tests, some summary information is

displayed about the test failure, and a debugging session is started. By default, Switchyard uses Python's built-in pdb debugger. At the very end of the output, a stack trace is shown and a debugger prompt is displayed:

Listing 3.4: Additional output from a test failure. Notice the error diagnosis in the output below, as well as how Switchyard invokes the debugger (pdb) at the point of failure.

```
************************************************************
Your code didn't crash, but a test failed.
************************************************************

This is the Switchyard equivalent of the blue screen of death.
As far as I can tell, here's what happened:

    Expected event:
        The Ethernet frame with a broadcast destination address
        should be forwarded out ports eth0 and eth2

    Failure observed:
        You called send_packet with an unexpected output port eth1.
        Here is what Switchyard expected: send_packet(s) Ethernet
        30:00:00:00:00:02->ff:ff:ff:ff:ff:ff IP | IPv4
        172.16.42.2->255.255.255.255 ICMP | ICMP EchoRequest 0 0 (0
        data bytes) out eth0 and Ethernet
        30:00:00:00:00:02->ff:ff:ff:ff:ff:ff IP | IPv4
        172.16.42.2->255.255.255.255 ICMP | ICMP EchoRequest 0 0 (0
        data bytes) out eth2.

You can rerun with the -v flag to include full dumps of packets that
may have caused errors. (By default, only relevant packet context may
be shown, not the full contents.)


I'm throwing you into the Python debugger (pdb) at the point of failure.
If you don't want pdb, use the --nopdb flag to avoid this fate.

> /Users/jsommers/Dropbox/src/switchyard/switchyard/llnettest.py(95)send_packet()
-> SwitchyardTestEvent.EVENT_OUTPUT, device=devname, packet=pkt)
> /Users/jsommers/Dropbox/src/switchyard/documentation/code/inout1.py(6)main()
-> net.send_packet(input_port, packet)
(Pdb)
```

Again, notice that the last couple lines show a (partial) stack trace. These lines can help a bit to understand the context of the error, but it is often helpful to show the source code around the failed code in light of the error diagnosis under "Failure observed", which says that we called `send_packet` with an unexpected output port. If we keep reading the diagnosis, we see that the packet was expected to be forwarded out two ports (eth0 and eth2), but was instead sent on eth1. Showing the source code can be accomplished with pdb's `list` command:

Listing 3.5: Output from pdb when listing the source code at the point of failure.

```
(Pdb) list
  8
  9         # alternatively, the above line could use indexing, although
 10         # readability suffers:
 11         #    recvdata[0], recvdata[2], recvdata[1]))
 12
 13  ->     net.send_packet(recvdata.input_port, recvdata.packet)
```

```
 14
 15         # likewise, the above line could be written using indexing
 16         # but, again, readability suffers:
 17         # net.send_packet(recvdata[1], recvdata[2])
[EOF]
(Pdb)
```

Between thinking about the observed failure and viewing the code, we might realize that we have foolishly sent the frame out the same interface on which it arrived.

3.4 Another example

To give a slightly different example, let's say that we're developing the code for a network hub, and because we love sheep, we decide to set every Ethernet source address to `ba:ba:ba:ba:ba:ba`. When we execute Switchyard in test mode (e.g., `swyard -t hubtests.py baaadhub.py`), we get the following output:

Listing 3.6: Test output for an example in which all Ethernet source addresses have been hijacked by sheep.

```
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


Failed:
    The Ethernet frame with a broadcast destination address
    should be forwarded out ports eth0 and eth2
        Expected event: send_packet(s) Ethernet
        30:00:00:00:00:02->ff:ff:ff:ff:ff:ff IP | IPv4
        172.16.42.2->255.255.255.255 ICMP | ICMP EchoRequest 0 0 (0
        data bytes) out eth0 and Ethernet
        30:00:00:00:00:02->ff:ff:ff:ff:ff:ff IP | IPv4
        172.16.42.2->255.255.255.255 ICMP | ICMP EchoRequest 0 0 (0
        data bytes) out eth2


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
2   Ethernet frame destined for 30:00:00:00:00:02 should be
    flooded out eth1 and eth2
3   An Ethernet frame from 30:00:00:00:00:02 to
    20:00:00:00:00:01 should arrive on eth1
4   Ethernet frame destined to 20:00:00:00:00:01 should be
    flooded out eth0 and eth2
5   An Ethernet frame should arrive on eth2 with destination
    address the same as eth2's MAC address
6   The hub should not do anything in response to a frame
    arriving with a destination address referring to the hub
    itself.


************************************************************
Your code didn't crash, but a test failed.
************************************************************
```

```
This is the Switchyard equivalent of the blue screen of death.
As far as I can tell, here's what happened:

    Expected event:
        The Ethernet frame with a broadcast destination address
        should be forwarded out ports eth0 and eth2

    Failure observed:
        You called send_packet and while the output port eth0 is ok,
        an exact match of packet contents failed.  In the Ethernet
        header, src is wrong (is ba:ba:ba:ba:ba:ba but should be
        30:00:00:00:00:02).

... output continues ...
```

In this case, we can see that the first section is basically the same as with the other erroneous code, but the failure description is different: Switchyard tells us that in the Ethernet header, the `src` attribute was wrong. If, at the `pdb` prompt, we type `list`, we see our wooly problem:

Listing 3.7: Pdb source code listing showing the point of test failure.

```
(Pdb) list
 28             else:
 29                 for intf in my_interfaces:
 30                     if dev != intf.name:
 31                         log_info ("Flooding packet {} to {}".format(packet, intf.name))
 32                         eth.src = 'ba:ba:ba:ba:ba:ba' # sheep!
 33  ->                     net.send_packet(intf, packet)
 34         net.shutdown()
[EOF]
(Pdb)
```

So, although the error diagnosis cannot generally state *why* a problem has happened, it can sometimes be quite specific about *what* has gone wrong. That, coupled with showing the source code context, can be very helpful for tracking down bugs. It might also be helpful to note that at the pdb prompt, you can inspect *any* variable in order to figure out what's happened, walk up and down the call stack and execute arbitrary Python statements in order to try to determine what has happened. Debuggers can be a little bit daunting, but they're incredibly helpful tools.

See also:

To learn more about pdb and the various commands and capabilities it has, refer to the Python library documentation (there's a section specifically on `pdb`). There are other debuggers out there with additional features, but `pdb` is *always* available with any Python distribution so it is worth acquainting yourself with it.

3.5 Even more verbose output

If you'd like even more verbose output, you can add the `-v` (verbose) and/or `-d` (debug) flags to `swyard`. The `-d` flag may be more trouble than it's worth since it enables all DEBUG-level log messages to be printed to the console. If you're really stuck trying to figure out what's going on, however, this may be useful.

3.6 If you don't like pdb

If you don't appreciate being dumped into the `pdb` debugger when something fails (maybe you're a cretin who really just likes `printf`-style debugging?), you can add the `--nopdb` flag to `swyard`. With the `--nopdb` option, Switchyard will print out information about test failure, but you'll go straight back to a command-line prompt.

If you'd like to use a debugger, but just not `pdb`, you can use the `--nohandle` (or `-e`) option to tell Switchyard not to trap any exceptions, but to let them be raised normally. You can then catch any exceptions using an alterative debugger. For example, if you'd like to use the `PuDB` debugger, you could invoke `swyard` as follows:

```
$ python3 -m pudb.run swyard --nohandle ...
```

where the ellipsis is replaced with other command-line arguments to `swyard`.

3.6.1 Debugging Switchyard code

When running Switchyard, especially in test mode, it is often very helpful to use the interactive Python debugger as you work out problems and figure things out. With the `import` of `switchyard.lib.userlib` you get a function named `debugger`. You can insert calls to the `debugger` function where ever you want to have an interactive debugger session start up. For example, we could create a simple program that starts up a debugger session when ever we receive a packet:

```
from switchyard.lib.userlib import *

def main(net):
    while True:
        try:
            timestamp,input_port,packet = net.recv_packet(timeout=1.0)
        except NoPackets:
            # timeout waiting for packet arrival
            continue
        except Shutdown:
            # we're done; bail out of while loop
            break

        # invoke the debugger every time we get here, which
        # should be for every packet we receive!
        debugger()
        hdrs = packet.num_headers()

    # before exiting our main function,
    # perform shutdown on network
    net.shutdown()
```

If we run the above program, we will stop at the line *after* the call to `debugger`:

Listing 3.8: When the debugger() call is added to a Switchyard program, execution is halted at the *next* line of code.

```
> /users/jsommers/dropbox/src/switchyard/documentation/code/enterdebugger.py(17)main()
-> hdrs = packet.num_headers()
(Pdb) list
 12                 break
 13
```

```
 14             # invoke the debugger every time we get here, which
 15             # should be for every packet we receive!
 16             debugger()
 17  ->         hdrs = packet.num_headers()
 18
 19         # before exiting our main function,
 20         # perform shutdown on network
 21         net.shutdown()
[EOF]
(Pdb)
```

Note: There are currently a couple limitations when entering `pdb` through a call to `debugger()`. First, if you attempt to exit `pdb` while the Switchyard program is still running, an exception from `pdb`'s base class (`Bdb`) will get raised. Thus, it may take a couple invocations of the `quit` command to actually exit. Second, only the `pdb` debugger may be invoked through a call to `debugger`.

As noted above, if there is a runtime error in your code, Switchyard will automatically dump you into the Python debugger (pdb) to see exactly where the program crashed and what may have caused it. You can use any Python commands to inspect variables, and try to understand the state of the program at the time of the crash.

3.7 Checking code coverage

If you want to check which lines of code are *covered* by one or more test scenarios, you can install and use the `coverage` package. This can be helpful for seeing which lines of your code are *not* being exercised by tests, and how you might focus additional testing effort.

To install:

```
$ pip3 install coverage
```

To gather code coverage information, you can invoke `swyard` using `coverage`. `coverage` appears to work best if you give the full path name of `swyard`, which is what the following command line will do (using backtick-substitution for the `which swyard` command). You can use command-line options to `swyard` as you normally would:

```
$ coverage run `which swyard` -v -d -t testscenario.py yourcode.py
```

Once you've created the coverage information you can display a report. The html report will nicely show exactly which lines of your code were executed during a test and which weren't. To avoid seeing coverage information for irrelevant files, you should explicitly tell `coverage` which files you want to include in the report.

```
$ coverage html --include yourcode.py
```

After running the above command, you can open the file `index.html` within the `htmlcov` folder. Clicking on a file name will show detailed coverage information.

CHAPTER

FOUR

TEST SCENARIO CREATION

Writing tests to determine whether a piece of code behaves as expected is an important part of the software development process. With Switchyard, it is possible to create a set of tests that verify whether a program attempts to receive packets when it should and sends the *right* packet(s) out the *right* ports. This section describes how to construct such tests.

A *test scenario* is Switchyard's term for a series of tests that verify a program's behavior. A test scenario is simply a Python source code file that includes a particular variable name (symbol) called `scenario`, which must refer to an instance of the class `TestScenario`. A `TestScenario` object contains the basic configuration for an imaginary network device along with an ordered series of *test expectations*. These expectations may be one of three types:

- that a particular packet should arrive on a particular interface/port,
- that a particular packet should be emitted out one or more ports, and
- that the user program should *time out* when calling `recv_packet` because no packets are available.

To start off, here is an example of an *empty* test scenario:

Listing 4.1: An empty test scenario.

```
from switchyard.lib.userlib import *

scenario = TestScenario("test example")
```

If we run `swyard` in test mode using this test description and *any* Switchyard program, here's the output we should see:

```
Results for test scenario test example: 0 passed, 0 failed, 0 pending


All tests passed!
```

Notice that in the above example code, we assigned the instance of the `TestScenario` class to a variable named `scenario`. An assignment to this variable name is **required**. If it is not found, you'll get an `ImportError` exception. Notice also that there's one parameter to `TestScenario`: this value can be any meaningful description of the test scenario.

There are two methods on `TestScenario` that are used to configure the test environment:

- `add_interface(name, macaddr, ipaddr=None, netmask=None, **kwargs)`

  This method adds an interface/port to an imaginary network device that is the subject of the test scenario. For example, if you are creating a test for an IP router and you want to verify that a packet received on one port is forwarded out another (different) port on the device, you will need to add *at*

least two interfaces. Arguments to the `add_interface` method are used to specify the interface's name (e.g., `en0`), its hardware Ethernet (MAC) address, and its (optional) IP address and netmask.

Two optional keyword arguments can also be given: `ifnum` can be used to explicitly specify the number (integer) associated with this interface, and `iftype` can be used to explicitly indicate the type of the interface. A value from the enumeration `InterfaceType` must be used, e.g., `Wired`, `Wireless`, `Loopback`, or `Unknown`. The type of an interface defaults to `InterfaceType.Unknown`.

- `add_file(filename, text)`

  It is sometimes necessary to make sure that certain text files are available during a test that a user program expects, e.g., a static forwarding table for an IP router. This method can be used to specify that a file with the name `filename` and with contents `text` should be written to the current directory when the test scenario is run.

There is one method that creates a new test expectation in the test scenario:

- `expect(expectation_object, description)`

  This method adds a new expected event to the test scenario. The first parameter must be an object of type `PacketInputEvent`, `PacketInputTimeoutEvent`, or `PacketOutputEvent` (each described below). The order in which expectations are added to a test scenario is critical: be certain that they're added in the right order for the test you want to accomplish!

  The description parameter is a short text description of what this test step is designed to accomplish. In `swyard` test output, this description is what is printed for each step in both the abbreviated and verbose output: make sure it is descriptive enough so that the purpose of the test can be easily understood. At the same time, try to keep the text short so that it isn't overwhelming to a reader.

The three *event* classes set up the specific expectations for each test, as described next.

- `PacketInputEvent(portname, packet, display=None, copyfromlastout=None)`

  Create an expectation that a particular packet will arrive and be received on a port named `portname`. The packet must be an instance of the Switchyard `Packet` class. The `portname` is just a string like `eth0`. This port/interface must have previously be configured in the test scenario using the method `add_interface` (see above).

  The `display` argument indicates whether a particular header in the packet should be emphasized on output when Switchyard shows test output to a user. By default, all headers are shown. If a test creator wants to ignore the Ethernet header but emphasize the IPv4 header, he/she could use the argument `display=IPv4`. That is, the argument is just the class name of the packet header to be emphasized.

  The `copyfromlastout` argument can be used to address the situation in which a test scenario author wants to construct an incoming packet (that will be received by `recv_packet`) which has the same values in some packet header fields as the most recent packet emitted. For example, when creating a protocol stack, an application (socket) program might emit a packet with a source port number assigned by the socket emulation module. The destination port number in an arriving packet needs to be the same as the packet that was previously emitted in order for it to be handed to the correct application program. Thus, the `copyfromlastout` can be used to copy one or more packet header attributes from the *last* emitted packet to header fields in an incoming packet.

  `copyfromlastout` can take a tuple of 5 elements: the interface/port name out which the packet was sent, a header class name and attribute to copy *from*, and a header class name and attribute to copy *to*. For example, if we wanted to copy the UDP source port value from the last packet emitted out port `en1` to the UDP destination port of the packet to be received, we could use the following:

```
PacketInputEvent('en1', pkt, copyfromlastout('en1', UDP, 'src', UDP, 'dst'))
```

  Note that we would need to have created a `Packet` object named `pkt` which included a UDP header for this example to work correctly.

- `PacketInputTimeoutEvent(timeout)`

  Create an expectation that the Switchyard user program will call `recv_packet` but *time out* prior to receiving anything. The timeout value is the number of seconds to wait within the test framework before raising the `NoPackets` exception in the user code. In order for this test expectation to pass, the user code must correctly handle the exception and must not emit a packet.

  To force a `NoPackets` exception, the timeout value given to this event must be greater than the timeout value used in a call to `recv_packet`. Note also that the test framework will pause for the *entire* duration of the given timeout. If a user program calls `net.recv_packet(timeout=1.0)` but the timeout given for a `PacketInputTimeoutEvent` is 5 seconds, the call to `recv_packet` will appear to have blocked for 5 seconds, not 1.

- `PacketOutputEvent(*args, display=None, exact=True, predicates=[], wildcard=[])`

  Create an expectation that the user program will emit packets out one or more ports/interfaces. The only required arguments are `args`, which must be an **even number** of arguments. For each pair of arguments given, the first is a port name (e.g., `en0`) and the second is a reference to a packet object. Normally, a test wishes to establish that the *same* packet has been emitted out multiple interfaces. To do that, you could simply write:

```
p = Packet()
# fill in some packet headers ...
PacketOutputEvent('en0', pkt, 'en1', pkt, 'en2', pkt)
```

  The above code expects that the same packet (named `pkt`) will be emitted out three interfaces (`en0`, `en1`, and `en2`).

  By default, the PacketOutputEvent class looks for an **exact** match between the reference packet supplied to PacketOutputEvent and the packet that the user code actually emits. In some cases, this isn't appropriate or even possible. For example, you may want to verify that packets are forwarded correctly using standard IP (longest prefix match) forwarding rules, but you may not know the payload contents of a packet because another test element may modify them. As another example, in IP forwarding you know that the TTL (time-to-live) should be decremented by one, but the specific value in an outgoing packet depends on the value on the incoming packet, which the test framework may not know in advance. To handle these situations, you can supply `exact`, `wildcard(s)`, and/or `predicate(s)` keyword arguments, as detailed below.

  - **Exact vs. subset matching**: Setting `exact` to `True` or `False` determines whether *all* packet header attributes are compared (`exact=True`) or whether a limited subset are compared (`exact=False`).

    The set of header fields that are compared when `exact=False` is specified are: Ethernet source and destination addresses, Ethernet ethertype field, Vlan vlanid and ethertype field, ARP target and sender protocol and hardware addresses (four fields), IPv4/IPv6 source and destination addresses and protocol, and TCP/UDP src/dst port numbers (or ICMP/ICMPv6 icmptype/icmpcode fields). Note that in subset matching no packet payloads are compared.

  - **Wildcard fields**: In addition to specifying the `exact` keyword parameter, it is possible to specify that some additional header fields should be *wildcarded*. That is, the wildcarded header fields are allowed to contain *any* value. Wildcards are specified using a tuple of two elements: a header class name and a field name.

    A single wildcard can be supplied (i.e., one 2-tuple) with the `wildcard` keyword parameter, or a *list* of 2-tuples can be supplied with the `wildcards` keyword. For example, the following line of code uses subset matching (`exact=False`) and one wildcard. For this example, assume that the packet `pkt` contains `Ethernet`, `IPv4`, and `UDP` headers:

```
PacketOutputEvent('en0', pkt, exact=False, wildcard=(IPv4, 'src'))
```

Note that for the above example, the only fields compared in the IPv4 header would be the destination address and protocol field (since other fields are already ignored with `exact=False`).

Here is another example that ignores source addresses in the Ethernet, IPv4 and UDP fields, leaving only two fields in the Ethernet header to be compared (dst and ethertype), two fields to be compared in the IPv4 header (dst and protocol) and one field in UDP (dst). Again, assume that the packet `pkt` contains `Ethernet`, `IPv4`, and `UDP` headers:

```
PacketOutputEvent('en0', pkt, exact=False, wildcards=[(Ethernet, 'src'), (IPv4, 'src'),␣
↪(UDP, 'src')])
```

Note: Switchyard previously allowed certain strings (modeled on the Openflow 1.0 specification) to be used to indicate wildcarded fields. These strings can *no longer be used* in the current version of Switchyard. To specify wildcarded fields, you **must** use the `(hdrclass, attribute)` syntax.

– **Predicate functions**: Lastly, predicate functions can be supplied to make *arbitrary* tests against packets. The `predicate` keyword argument can take a single `lambda` function in the form of a string, and the `predicates` keyword argument can take a *list* of `lambda` functions, each as strings. Each lambda given must take a single argument (the packet object to be inspected) and must yield a boolean value. (Note that internally, each lambda definition is `eval`'ed by Switchyard.)

Here is one example that checks whether the IPv4 ttl field is between 32 and 34, inclusive. Note that this line of code contains a *single* predicate function as a string:

```
PacketOutputEvent('en1', pkt, exact=False, predicate='''lambda p: p.has_header(IPv4) and␣
↪32 <= p[IPv4].ttl <= 34''')
```

To provide multiple predicates, just use the `predicates` (plural) keyword and provide a list of lambdas-as-strings.

4.1 Test scenario examples

First, here is an example of a test scenario in which a packet is constructed and is expected to be received on port `eth1`, then sent back out the same port, unmodified. Notice in the example that the name `scenario` is *required*.

Listing 4.2: A test scenario in which a packet is received then sent back out the same port.

```
from switchyard.lib.userlib import *

scenario = TestScenario("in/out test scenario example")

# only one interface on this imaginary device
scenario.add_interface('eth0', 'ab:cd:ef:ab:cd:ef', '1.2.3.4', '255.255.0.0',
    iftype=InterfaceType.Wired)

# construct a packet to be received
p = Ethernet(src="00:11:22:33:44:55", dst="66:55:44:33:22:11") + \
    IPv4(src="1.1.1.1", dst="2.2.2.2", protocol=IPProtocol.UDP) + \
    UDP(src=5555, dst=8888) + b'some payload'

# expect that the packet is received
scenario.expect(PacketInputEvent('eth0', p),
    "A udp packet should arrive on eth0")
```

```
# and expect that the packet is sent right back out
scenario.expect(PacketOutputEvent('eth0', p, exact=True),
    "The udp packet should be emitted back out eth0")
```

Here is an additional example with a bit more complexity. The context for this example might be that we are implementing an IPv4 router. First, notice that we include in the scenario a static forwarding table file (`forwarding_table.txt`) to be written out when the scenario is executed. We construct a packet destined to a particular IP address and create an expectation that it arrives on port `eth0`. We then construct an expectation that the packet should be forwarded out port `eth2` (note that according to the forwarding table, any packets destined to 2.0.0.0/8 should be forwarded out that port). We also include a predicate function to test that the IPv4 ttl is decremented by 1. Note that if we did not include this predicate, *any* ttl value would be accepted since we have specified `exact=False`. Note also that if we had set `exact=True` we would almost certainly need to wildcard several fields, e.g., checksums in the IPv4 and UDP headers, and would still need to include a predicate to check that ttl has been properly decremented. Furthermore, if we were writing a test scenario for an IP router, we would also want to include expectations that the correct ARP messages were sent in order to obtain the hardware address corresponding to the next hop IP address.

Listing 4.3: A simplified IP forwarding test scenario.

```
from switchyard.lib.userlib import *

scenario = TestScenario("packet forwarding example")

# three interfaces on this device
scenario.add_interface('eth0', 'ab:cd:ef:ab:cd:ef', '1.2.3.4', '255.255.0.0')
scenario.add_interface('eth1', '00:11:22:ab:cd:ef', '5.6.7.8', '255.255.0.0')
scenario.add_interface('eth2', 'ab:cd:ef:00:11:22', '9.10.11.12', '255.255.255.0')

# add a forwarding table file to be written out when the test
# scenario is executed
scenario.add_file('forwarding_table.txt', '''
# network   subnet-mask   next-hop      port
2.0.0.0     255.0.0.0     9.10.11.13    eth2
3.0.0.0     255.255.0.0   5.6.100.200   eth1
''')


# construct a packet to be received
p = Ethernet(src="00:11:22:33:44:55", dst="66:55:44:33:22:11") + \
    IPv4(src="1.1.1.1", dst="2.2.2.2", protocol=IPProtocol.UDP, ttl=61) + \
    UDP(src=5555, dst=8888) + b'some payload'

# expect that the packet is received
scenario.expect(PacketInputEvent('eth0', p),
    "A udp packet destined to 2.2.2.2 arrives on port eth0")

# and subsequently forwarded out the correct port; employ
# subset (exact=False) matching, along with a check that the
# IPv4 TTL was decremented exactly by 1.
scenario.expect(PacketOutputEvent('eth2', p, exact=False,
    predicate='''lambda pkt: pkt.has_header(IPv4) and pkt[IPv4].ttl == 60'''),
    "The udp packet destined to 2.2.2.2 should be forwarded out port eth2, with an appropriately␣
↪decremented TTL.")
```

4.2 Compiling a test scenario

A test scenario can be run *directly* with `swyard` or it can be *compiled* into a form that can be distributed without giving away the code which was used to construct it. Compiled test scenario files are, by default, given a `.srpy` extension; uncompiled test scenarios should just be regular Python (`.py`) files.

To compile a test scenario, you can simply invoke `swyard` with the `-c` flag, as follows:

```
swyard -c code/testscenario2.py
```

The output from this command should be a new file named `code/testscenario2.srpy` containing the obfuscated test scenario. This file can be used as the argument to the `-t` option when later running a Switchyard program against those tests.

Note: Note that if a scenario is *compiled* using a different version of Python than the one used to *run* a test scenario (especially a different major version, e.g., 3.4 vs. 3.5), you may get some mysterious errors. The errors are due to the fact that serialized representations of Python objects may change from one version to the next; if there are any changes, then the version used to run the test cannot correctly deserialize the various objects stored in the test scenario.

CHAPTER

FIVE

RUNNING IN A "LIVE" ENVIRONMENT

Switchyard programs can be either run in an isolated *test environment*, as described above, or on a *live* host operating system. Switchyard currently supports Linux and macOS hosts for live execution.

Note: Switchyard uses the `libpcap` library for receiving and sending packets, which generally requires *root* privileges. Although hosts can be configured so that root isn't required for using `libpcap`, this documentation does not include instructions on how to do so. The discussion below assumes that you are gaining root privileges by using the `sudo` (i.e., "do this as superuser") program. Contrary to popular belief, `sudo` cannot make you a sandwich.

5.1 Basic command-line recipe

The basic recipe for running Switchyard on a live host is pretty simple. If we wanted to run the `sniff.py` Switchyard program (available in the `examples` folder in the Switchyard github repository) and use *all* available network interfaces on the system, we could do the following:

```
$ sudo swyard sniff.py
```

Again, note that the above line uses `sudo` to gain the necessary privileges to be able to send and receive "live" packets on a host.

Note: If you can an error when attempting to run `swyard` with `sudo` such as this:

```
sudo: swyard: command not found
```

you will need to either create a shell script which activates your Python virtual environment and run that script with `sudo`, or run `swyard` from a root shell (e.g., by running `sudo -s`. If doing the latter, you will still need to activate the Python virtual environment once you start the root shell, after which you can run `swyard` as normal. If using Switchyard in Mininet, in any shell you open (e.g., using the `xterm` command, which opens a root shell on a virtual host in Mininet) you'll need to activate the Python virtual environment prior to running `swyard`.

The `sniff.py` program will simply print out the contents of any packet received on *any* interface while the program runs. To stop the program, type `Control+c`.

Here's an example of what output from running `sniff.py` might look like. Note that the following example was run on a macOS host and that the text times/dates have been changed:

Listing 5.1: Example of Switchyard output from running in a live environment on a macOS host.

```
00:00:56 2016/12/00     INFO Enabling pf: No ALTQ support in kernel; ALTQ related functions␣
↪disabled; pf enabled; Token : 15170097737539790927
00:00:56 2016/12/00     INFO Using network devices: en1 en0 en2
00:00:56 2016/12/00     INFO My interfaces: ['en0', 'en1', 'en2']
00:00:56 2016/12/00     INFO 1482563936.430: en0 Ethernet a4:71:74:49:e2:e6->ac:bc:32:c2:b6:59 IP␣
↪| IPv4 104.84.41.100->192.168.0.102 TCP | TCP 443->51094 (A 1772379675:466295739) |␣
↪RawPacketContents (1448 bytes) b'\x17\x03\x03\x0c-\xc5\xeap\xd1L'...
00:00:56 2016/12/00     INFO 1482563936.430: en0 Ethernet a4:71:74:49:e2:e6->ac:bc:32:c2:b6:59 IP␣
↪| IPv4 104.84.41.100->192.168.0.102 TCP | TCP 443->51094 (A 1772381123:466295739) |␣
↪RawPacketContents (1448 bytes) b'\xca5K\xfb\x88\x01\xec\xb4\xf0\x84'...
00:00:56 2016/12/00     INFO 1482563936.430: en0 Ethernet a4:71:74:49:e2:e6->ac:bc:32:c2:b6:59 IP␣
↪| IPv4 104.84.41.100->192.168.0.102 TCP | TCP 443->51094 (PA 1772382571:466295739) |␣
↪RawPacketContents (226 bytes) b'\xb1\x9d\xad8g]\xc3\xech\x9e'...

... (more packets, removed for this example)

^C
00:00:58 2016/12/00     INFO Releasing pf: No ALTQ support in kernel; ALTQ related functions␣
↪disabled; disable request successful. 1 more pf enable reference(s) remaining, pf still enabled.
```

Note in particular a few things about the above example:

- First, when started in a live setting, Switchyard *saves* then *clears* any current host firewall settings. The saved firewall settings are restored when Switchyard exits (see the final log line, above).

  The default behavior of Switchyard is to *block all traffic*. This behavior may be undesirable in different situations and can be changed through the `swyard` command line option `-f` or `--firewall`, as described below.

  Switchyard's manipulation of the host operating system firewall is intended to prevent the host from receiving any traffic that should be the sole domain of Switchyard. For example, if you are creating a Switchyard-based IP router, you want Switchyard, not the host, to be responsible for receiving and forwarding traffic. As another example, if you are implementing a protocol stack for a particular UDP-based application, you will want to prevent the host from receiving any of that UDP traffic.

  Note that on macOS Switchyard configures host firewall settings using `pfctl` and on Linux Switchyard uses `iptables`.

- By default, Switchyard finds and uses all interfaces on the host that are (1) determined to be "up" (according to libpcap), and (2) *not* a localhost interface. In the above example run, Switchyard finds and uses three interfaces (`en0`, `en1`, and `en2`).

- The above example shows three packets that were observed by Switchyard, each arriving on interface `en0`. Notice that the three packets each contain Ethernet, IPv4 and TCP packet headers, as well as payload (in the form of `RawPacketContents` objects at the end of each packet).

Here is an example of running the Switchyard example `sniff.py` program on a Linux host (note again that the text times/dates have been changed):

Listing 5.2: Example of Switchyard output from running in a live environment on a Linux host.

```
00:00:11 2016/12/00     INFO Saving iptables state and installing switchyard rules
00:00:11 2016/12/00     INFO Using network devices: enp0s3
00:00:11 2016/12/00     INFO My interfaces: ['enp0s3']
00:00:15 2016/12/00     INFO 1482564855.115: enp0s3 Ethernet 08:00:27:bb:27:89->01:00:5e:00:00:fb␣
↪IP | IPv4 10.0.2.15->224.0.0.251 UDP | UDP 5353->5353 | RawPacketContents (45 bytes) b
↪'\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00'...
```

```
00:00:16 2016/12/00     INFO 1482564856.172: enp0s3 Ethernet 08:00:27:bb:27:89->33:33:00:00:00:fb␣
↪IPv6 | IPv6 fe80::a00:27ff:febb:2789->ff02::fb UDP | UDP 5353->5353 | RawPacketContents (45␣
↪bytes) b'\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00'...

... (more packets, removed for this example)

^C
00:00:23 2016/12/00     INFO Restoring saved iptables state
```

Comparing the above output to the earlier macOS output, observe that:

- The firewall save/restore log lines (first and last) are somewhat different, reflecting the fact that `iptables` is used on Linux instead of `pf`.
- There is one interface found and used by Switchyard: `enp0s3`.
- Two packets are included in the output above: an IPv4 UDP packet and an IPv6 UDP packet.

As with running Switchyard in a test environment, you may wish to use the `-v` and/or `-d` options to increase Switchyard's output verbosity or to include debugging messages, respectively.

5.2 Including or excluding particular interfaces

When running Switchyard in a virtual machine environment such as on a Mininet container host, it is often the case that you want Switchyard to "take over" all available network interfaces on the host. When running Switchyard in other environments, however, you may want to restrict the interfaces that it uses. You may even want Switchyard to use the localhost interface (typically named `lo0` or `lo`). There are two command-line options that can be used for these purposes.

`-i` `<interface-name>`
: Explicitly *include* the given interface for use by Switchyard. This option can be used more than once to include more than one interface.

  If this option is given, *only* the interfaces specified by `-i` options will be used by Switchyard. If no `-i` option is specified, Switchyard uses all available interfaces *except* the localhost interface.

  To use a localhost interface, you must explicitly include it using this option. If you explicitly include the localhost interface, you can still explicitly include other interfaces.

`-x` `<interface-name>`
: Explicitly *exclude* the given interface for use by Switchyard. This option can be used more than once to exclude more than one interface.

  Switchyard's behavior with this option is to first discover *all* interfaces available on the host, then to remove any specified by `-x`.

Note that given the semantics described above, it generally makes sense only to specify *one* of `-i` or `-x`.

5.3 Firewall options

As noted above, Switchyard's default behavior is to prevent the host operating system from receiving any traffic while Switchyard is running. This may be undesirable in certain situations, and the `-f` or `--firewall` options to `swyard` are available to change this behavior.

The `-f` and `--firewall` options accept a single rule as a parameter (which in many cases needs to be quoted in the shell). The rule syntax is `proto[:port]`, where the `[:port]` part is optional and `proto` may be one

of `tcp`, `udp`, `icmp`, `none` or `all`. If `all` is specified, the port part should not be included; `all` will block *all* traffic on the interfaces used by Switchyard. If `none` is specified, again, no port should be specified; `none` will cause *no rules to be installed* to block traffic. Here are some examples:

`tcp` Block the host from receiving all TCP traffic

`tcp:8000` Block the host from receiving TCP traffic on port 8000

`icmp` Block the host from receiving all ICMP traffic

`udp:4567` Block the host from receiving UDP traffic on port 4567

`none` Do not block any traffic.

`all` Block the host from receiving all traffic. This is the default behavior.

If the `-v` (verbose) option is given to `swyard`, the host firewall module will print (to the log) firewall settings that have been enabled. Here are two examples from running `swyard` in a live environment (on macOS with the `pf` firewall). First, an example showing Switchyard blocking *all* traffic on two interfaces:

Listing 5.3: Running Switchyard in a live environment (macOS) with -v flag: notice log line indicating firewall rules installed (2nd line, 2 rules).

```
$ sudo swyard -i lo0 -i en0 -v sniff.py
11:39:58 2016/12/00     INFO Enabling pf: No ALTQ support in kernel; ALTQ related functions␣
↪disabled; pf enabled; Token : 16107925605825483691;
11:39:58 2016/12/00     INFO Rules installed: block drop on en0 all
block drop on lo0 all
11:39:58 2016/12/00     INFO Using network devices: en0 lo0
11:39:58 2016/12/00     INFO My interfaces: ['en0', 'lo0']
^C11:40:00 2016/12/00     INFO Releasing pf: No ALTQ support in kernel; ALTQ related functions␣
↪disabled; disable request successful. 4 more pf enable reference(s) remaining, pf still enabled.;
```

Here is an example showing Switchyard blocking all ICMP, all TCP, and UDP port 8888:

Listing 5.4: Running Switchyard in a live environment (macOS) with -v flag: notice log line indicating firewall rules installed (2nd line, 3 rules).

```
$ sudo swyard -i lo0 --firewall icmp --firewall tcp --firewall 'udp:8888' -v sniff.py
11:43:46 2016/12/00     INFO Enabling pf: No ALTQ support in kernel; ALTQ related functions␣
↪disabled; pf enabled; Token : 16107925605472991531;
11:43:46 2016/12/00     INFO Rules installed: block drop on lo0 proto icmp all
block drop on lo0 proto tcp all
block drop on lo0 proto udp from any port = 8888 to any port = 8888
11:43:46 2016/12/00     INFO Using network devices: lo0
11:43:46 2016/12/00     INFO My interfaces: ['lo0']
^C11:43:48 2016/12/00     INFO Releasing pf: No ALTQ support in kernel; ALTQ related functions␣
↪disabled; disable request successful. 4 more pf enable reference(s) remaining, pf still enabled.;
```

And finally, the same example as previous, but on Linux with iptables:

Listing 5.5: Running Switchyard in a live environment (Linux) with -v flag: notice log line indicating firewall rules installed (2nd line, 3 rules).

```
# swyard -v sniff.py --firewall icmp --firewall udp:8888 --firewall tcp
19:53:42 2016/12/00     INFO Saving iptables state and installing switchyard rules
19:53:42 2016/12/00     INFO Rules installed: Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DROP       icmp --  0.0.0.0/0            0.0.0.0/0
DROP       udp  --  0.0.0.0/0            0.0.0.0/0           udp dpt:8888
DROP       tcp  --  0.0.0.0/0            0.0.0.0/0
```

```
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
19:53:42 2016/12/00     INFO Using network devices: enp0s3
19:53:42 2016/12/00     INFO My interfaces: ['enp0s3']
^C19:53:45 2016/12/00     INFO Restoring saved iptables state
```

Note: When using a loopback interface, there are a couple things to be aware of. First, while Switchyard normally uses `libpcap` for sending and receiving packets, a *raw socket* is used for sending packets on the localhost interface. This is done due to limitations on some operating systems, notably Linux. Receiving packets is still done with `libpcap`, though on different operating systems you may observe that packets are encapsulated differently. In particular, on Linux, an `Ethernet` header with zeroed addresses is used, while on macOS the BSD Null header is used, which just consists of a protocol number (i.e., the ethertype value normally found in the Ethernet header).

CHAPTER

SIX

ADVANCED API TOPICS

This section introduces two additional, and slightly advanced topics related to Switchyard APIs: creating new packet header types and using Switchyard's application-layer socket emulation capabilities.

6.1 Creating new packet header types

For some Switchyard programs, it can be useful to create new packet header types. For example, say you want to implement a simplified dynamic routing protocol within an IP router. You might want to be able to create a new packet header for your routing protocol, and have those packet headers integrate well with the existing Switchyard `Packet` class. Similarly, say you want to implement a simplified Ethernet spanning tree protocol: being able to create a new packet header for carrying spanning tree information would be helpful.

Before discussing how to create a new packet header class that integrates well with the rest of Switchyard, it is important to note that it is not strictly *required* to create a new packet header class for either of the above example projects. Instead, you could use the existing `RawPacketContents` header, which has one attribute (`data`), a `bytes` object. To use a `RawPacketContents` header, you would need to handle all *packing* ("serialization") and *unpacking* ("deserialization") of header fields to and from the bytes object explicitly in your code. While this approach "works", it leads to a less cohesive and encapsulated design and to code that may be a bit more difficult to debug because it is not well-integrated into Switchyard.

If you want to work with Switchyard's packet header and packet classes, there are two main steps to take:

- First, create a new class that derives from `PacketHeaderBase`. There are two required methods (`to_bytes()` and `from_bytes()`) that you'll need to write, and some other things to be aware of when writing this class.
- Second, some configuration to the packet header class that appears *before* your header in a normal packet needs to be done. This is just a matter of a couple method calls to do the configuration.

These steps are described below along with short examples and a longer (full) example follows.

6.1.1 Creating a new packet header class

As mentioned above, to create a new packet header class you must create a class that derives from `PacketHeaderBase`. There are two required methods to implement:

`to_bytes()` This method returns a serialized packet header in the form of a `bytes` object. One of the easiest ways to "pack" a set of values into a `bytes` object is to use Python's `struct` module (refer to the Python library documentation for details). The examples in this section use `struct`.

`from_bytes(raw)` This method accepts a bytes object as a parameter and returns a `bytes` object. It populates attributes in the packet header by unpacking the `bytes` object. The method should raise an exception

if there aren't enough bytes to fully reconstruct the packet header. Any part of the `bytes` object passed as a parameter that *aren't* used (i.e., there are more bytes passed in to the method than are necessary to reconstruct the header) should be returned by the method. As with the `to_bytes()` method, Python's `struct` module is useful for performing the unpacking.

There is one restriction when implementing a new packet header class:

- The `__init__` method should only take *optional* parameters. Switchyard assumes that a packet header object can be constructed which assigns attributes to reasonable default values, thus no explicit initialization parameters can be required by the constructor (`__init__`). Moreover, for compatibility with keyword-style attribute assignment in packet header classes, a `kwargs` parameter should be included and passed to the base class initialization method call and this call to the base class must come **last** in the `__init__` method.

Below is an example of a new packet header called `UDPPing` that contains a single attribute: `sequence`. This packet header is designed to be included in a packet following a `UDP` header. Besides implementing an `__init__` method (which optionally accepts an initial sequence value) and the two required methods, there are property getter and setter methods for `sequence` and a string conversion magic method. Note that we've decided to store the sequence value as a network-byte-order (big endian) unsigned 16 bit value (this is what the `!H` signifies for `_PACKFMT`: refer to the `struct` Python library documentation):

```
from switchyard.lib.userlib import *
import struct

class UDPPing(PacketHeaderBase):
    _PACKFMT = "!H"

    def __init__(self, seq=0, **kwargs):
        self._sequence = int(seq)
        PacketHeaderBase.__init__(self, **kwargs)

    def to_bytes(self):
        raw = struct.pack(self._PACKFMT, self._sequence)
        return raw

    def from_bytes(self, raw):
        packsize = struct.calcsize(self._PACKFMT)
        if len(raw) < packsize:
            raise ValueError("Not enough bytes to unpack UDPPing")
        attrs = struct.unpack(self._PACKFMT, raw[:packsize])
        self.sequence = attrs[0]
        return raw[packsize:]

    @property
    def sequence(self):
        return self._sequence

    @sequence.setter
    def sequence(self, value):
        self._sequence = int(value)

    def __str__(self):
        return "{} seq: {}".format(self.__class__.__name__, self.sequence)
```

Given the way the `UDPPing` packet header class has been defined, we can either set the `sequence` explicitly with the property setter, pass a value into the `__init__` method, or use keyword syntax:

```
>>> up1 = UDPPing()
>>> print(up1)
UDPPing seq: 0
>>> up2 = UDPPing()
>>> up2.sequence = 13
>>> print(up2)
UDPPing seq: 13
>>> up3 = UDPPing(sequence=42)
>>> print(up3)
UDPPing seq: 0
```

If we now create a full `Packet` object, we might do something like the following. Note that our code both *serializes* and *deserializes* the packet. We do this to test (at least in a limited way) that our `to_bytes()` and `from_bytes()` methods work as expected. Here is the code:

```
UDP_PING_PORT = 12345
pkt = Ethernet(src="11:22:11:22:11:22",
               dst="22:33:22:33:22:33") + \
      IPv4(src="1.2.3.4", dst="5.6.7.8",
           protocol=IPProtocol.UDP, ttl=64) + \
      UDP(src=55555, dst=UDP_PING_PORT) + \
      UDPPing(42)
print("Before serialize/deserialize:", pkt)
xbytes = pkt.to_bytes()
reanimated_pkt = Packet(raw=xbytes)
print("After deserialization:", reanimated_pkt)
```

And here is the output:

```
Before serialize/deserialize: Ethernet 11:22:11:22:11:22->22:33:22:33:22:33 IP | IPv4 1.2.3.4->5.6.
↪7.8 UDP | UDP 55555->12345 | UDPPing seq: 42
After deserialization: Ethernet 11:22:11:22:11:22->22:33:22:33:22:33 IP | IPv4 1.2.3.4->5.6.7.8␣
↪UDP | UDP 55555->12345 | RawPacketContents (2 bytes) b'\x00*'
```

Notice that the first line of output shows the full packet as we expect, including the final `UDPPing` header. The next line to follow, however, shows that the packet has been reconstructed with the final header as `RawPacketContents`, not `UDPPing`. What happened?

6.1.2 Configuring the lower-layer header class

What happened in the above example is that Switchyard does not have enough information to know that the bytes that follow the UDP header should be interpreted as the contents of a `UDPPing` packet. It is possible, however, to give this information to Switchyard.

Switchyard assumes that there exists one attribute in a packet header that can be used to determine how to map *values* of that attribute to a *packet header class*. Not surprisingly, these mappings are stored in the form of a Python dictionary. For example, by default the `Ethernet` class is configured to use the value of the `ethertype` attribute as a lookup *key* to determine the type of the packet header that follows. It contains a few initial mappings, including a mapping from `EtherType.IP` to `IPv4`. Similarly, the `IPv4` class uses values in the `protocol` attribute as keys to look up the packet header type that should come next.

Switchyard contains methods to make it possible to change the *attribute* on which lookups are performed, to *add* new mappings from a value on the mapped attribute to a packet header class, and to *completely (re)initialize* the mappings from attribute values to packet header classes. Noting that one should, of course, use care when modifying any existing mappings or when modifying the attribute on which mappings are performed, here are the three *class* methods available on `PacketHeaderBase`-derived classes:

set_next_header_class_key(attr) This method is used to specify the *attribute* on which lookups to determine the next header class should be performed. Switchyard-provided header classes contain sensible defaults for this value. For example, with `Ethernet` and `Vlan` this attribute is preconfigured as `ethertype`, for `IPv4` this attribute is configured as `protocol`, and for `IPv6` it is `nextheader`. There is no default configuration set for `UDP` or `TCP`, but the natural choice would be `dst` (i.e., to use the destination port as the key). Most other headers are configured with the empty string, indicating that no "next header" is assumed by Switchyard. In that case, Switchyard will construct a `RawPacketHeader` object containing the remaining bytes.

add_next_header_class(attr, hdrcls) This method is used to add a new attribute value-header class mapping to the next header mapping dictionary.

set_next_header_map(mapdict) This method can be used to replace any previous dictionary with a new one. Switchyard-provided header classes are configured with sensible defaults. Use with care, since a replacement of a next header class mapping in a highly dependend-upon header class (e.g, `IPv4`) will likely break lots of things.

Note: A key limitation of Switchyard, currently, is that arbitrary values for core protocol number enumerations (in particular, `EtherType` and `IPProtocol`) cannot be dynamically added and/or modified because Python's `enum` types are constant once created. This makes it impossible, at present, to use *arbitrary* protocol numbers for new layer 3 or 4 protocols and packet header types. This will be changed in a future version of Switchyard. In the meantime, a workaround is to use an existing protocol number which is not used in the next header map. For example, if you are implementing a routing protocol on top of IPv4, you could use `IPProtocol.OSPF` as the protocol number for your (non-OSPF) protocol since Switchyard does not have any current mapping between that protocol number and a packet header class.

Building on the previous example with `UDPPing`, if we add *two* lines of code to specify that the destination port should be used as a key to look up the correct next header in a packet, and to *register* a particular UDP destination port as being associated with the `UDPPing` protocol, the final couple bytes can get properly interpreted and deserialized into the right packet header (notice the first two lines of code, which are the *only* differences with the previous example):

```
UDP.add_next_header_class(UDP_PING_PORT, UDPPing)
UDP.set_next_header_class_key('dst')
pkt = Ethernet(src="11:22:11:22:11:22",
               dst="22:33:22:33:22:33") + \
      IPv4(src="1.2.3.4", dst="5.6.7.8",
           protocol=IPProtocol.UDP, ttl=64) + \
      UDP(src=55555, dst=UDP_PING_PORT) + \
      UDPPing(sequence=13)
print("Before serialize/deserialize:", pkt)
xbytes = pkt.to_bytes()
reanimated_pkt = Packet(raw=xbytes)
print("After deserialization:", reanimated_pkt)
```

Here is the output, showing

```
Before serialize/deserialize: Ethernet 11:22:11:22:11:22->22:33:22:33:22:33 IP | IPv4 1.2.3.4->5.6.
↪7.8 UDP | UDP 55555->12345 | UDPPing seq: 13
After deserialization: Ethernet 11:22:11:22:11:22->22:33:22:33:22:33 IP | IPv4 1.2.3.4->5.6.7.8␣
↪UDP | UDP 55555->12345 | UDPPing seq: 13
```

6.1.3 One more example

Here is one additional example. Say that we want to implement a simplified Ethernet spanning tree protocol and want to create a packet header that includes an identifier for the root note and an integer value which indicates the number of hops to the root. We could do the following:

```
from switchyard.lib.userlib import *
import struct

class SpanningTreeMessage(PacketHeaderBase):
    _PACKFMT = "6sxB"

    def __init__(self, root="00:00:00:00:00:00", **kwargs):
        self._root = EthAddr(root)
        self._hops_to_root = 0
        PacketHeaderBase.__init__(self, **kwargs)

    def to_bytes(self):
        raw = struct.pack(self._PACKFMT, self._root.raw, self._hops_to_root)
        return raw

    def from_bytes(self, raw):
        packsize = struct.calcsize(self._PACKFMT)
        if len(raw) < packsize:
            raise ValueError("Not enough bytes to unpack SpanningTreeMessage")
        xroot,xhops = struct.unpack(self._PACKFMT, raw[:packsize])
        self._root = EthAddr(xroot)
        self.hops_to_root = xhops
        return raw[packsize:]

    @property
    def hops_to_root(self):
        return self._hops_to_root

    @hops_to_root.setter
    def hops_to_root(self, value):
        self._hops_to_root = int(value)

    @property
    def root(self):
        return self._root

    def __str__(self):
        return "{} (root: {}, hops-to-root: {})".format(
            self.__class__.__name__, self.root, self.hops_to_root)
```

Here is some example code for how we might use this class. Note that since we are creating a protocol header that should follow the `Ethernet` header, we must (due to a current limitation with Switchyard) use an existing `ethertype` value. We are reusing the value `EtherType.SLOW` for no particular reason other than it is presently unused by Switchyard:

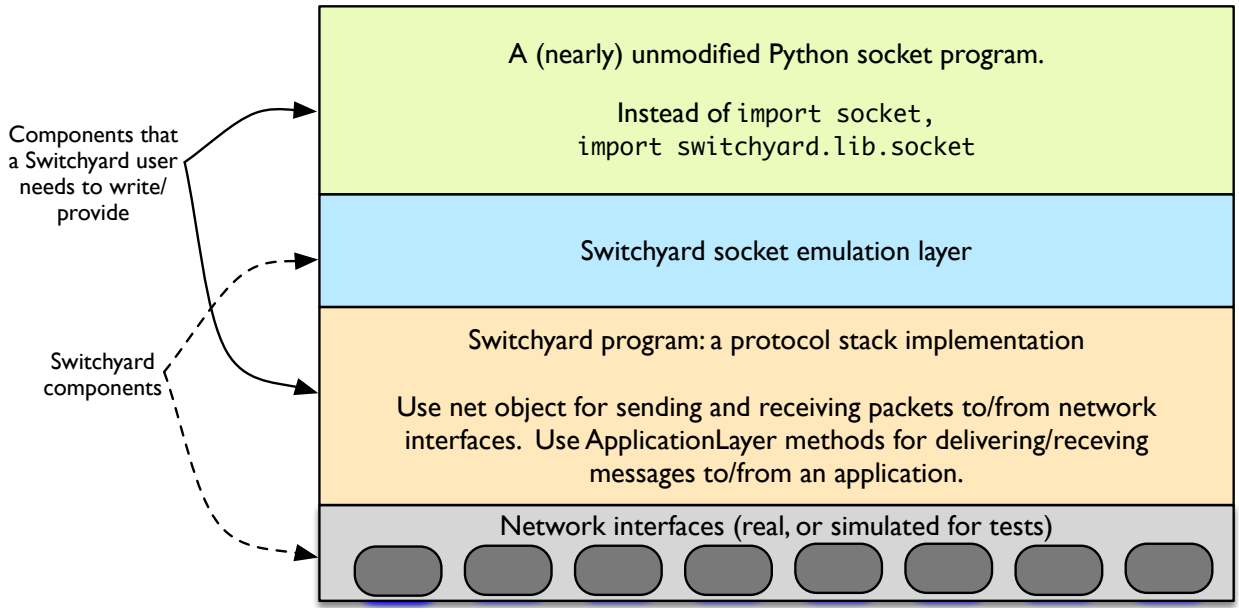
```
spm = SpanningTreeMessage("00:11:22:33:44:55", hops_to_root=1)
print(spm)

Ethernet.add_next_header_class(EtherType.SLOW, SpanningTreeMessage)
pkt = Ethernet(src="11:22:11:22:11:22",
               dst="22:33:22:33:22:33",
               ethertype=EtherType.SLOW) + spm
```

```
print(pkt)
xbytes = pkt.to_bytes()
p = Packet(raw=xbytes)
print(p)
```

6.2 Application layer socket emulation and creating full protocol stacks

It is possible within Switchyard to implement a program that resembles a full end-host protocol stack. The protocol stack can be used along with Switchyard's socket *emulation* library to execute nearly unmodified Python UDP socket programs. In this section, we discuss (1) additional API calls used to receive messages "down" from socket applications as well as deliver messages "up" to socket applications, (2) usage of and limitations with Switchyard's socket emulation library, and (3) additional command-line options with `swyard` for executing a socket application along with a protocol stack program.

A general picture of using Switchyard to execute a protocol stack *and* a socket application is shown below. Note that the figure shows two components that are provided (or controlled) by Switchyard, and two components that must be written or provided by a user of Switchyard.



6.2.1 API calls for delivering/receiving messages to/from applications

To deliver messages to or receive messages from a socket application, a Switchyard user must use two static methods on the `ApplicationLayer` class. These methods are similar in many ways to the two methods on the *net* object used to send and receive packets. The application-related methods are:

`ApplicationLayer.send_to_app(proto, local_addr, remote_addr, data)` This method is used to pass a message received on the network up to an application. The `proto` parameter is the IP protocol number of the packet from which the data was received. `local_addr` and `remote_addr` are 2-tuples consisting of an IP address and port. This method returns a boolean value: if there is a socket associated with the address information given, True is returned. Otherwise, False is returned.

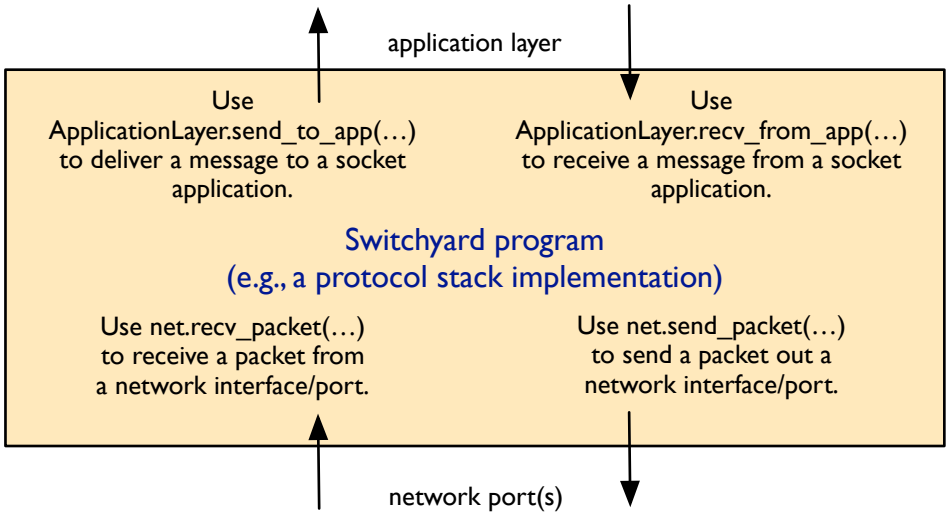
Note that if there is no socket associated with the address information given, a log warning is also emitted.

`ApplicationLayer.recv_from_app(timeout=None)` This method is used to receive an application message to be sent on the network. It takes an optional timeout argument which indicates the number of seconds to wait until giving up and raising a `NoPackets` exception. This exception is a bit of a misnomer here, but it is used for consistency with `net.recv_packet()`. If `None` is passed as a timeout value, this method will block until a message is available.

If a message is available, this method returns two items in the form of a tuple: a *flow address* and the data to be sent. The *flow address* consists of 5 items in the form of a tuple: the IP protocol value, a remote IP address and port, and the local IP address and port.

Note: if an application socket is *unbound*, the local IP address will be `0.0.0.0`. The protocol stack implementation is responsible for using a valid IP address in any outgoing packet (specifically, it should use the address assigned to the interface out which the packet is emitted).

In sum, there are 4 API calls that must be used to move packets and data through a protocol stack implementation, as shown in the figure below.



Using a similar pattern as with a "regular" Switchyard program, it is possible to service both of the incoming data channels (i.e., either packets received from a network port, or messages received from an application), as follows:

```
from switchyard.lib.userlib import *

class ProtocolStack(object):
    def __init__(self, net):
        self._net = net

    def handle_app_data(self, appdata):
        # do something to handle application data here, likely
        # resulting in an eventual call to self._net.send_packet()

    def handle_network_data(self, netdata):
        # do something with network data here, likely resulting
        # in an eventual call to ApplicationLayer.send_to_app()

    def main_loop(self):
```

```
        while True:
            appdata = None
            try:
                appdata = ApplicationLayer.recv_from_app(timeout=0.1)
            except NoPackets:
                pass
            except Shutdown:
                break
            if appdata is not None:
                handle_app_data(net, intf, appdata)

            netdata = None
            try:
                netdata = net.recv_packet(timeout=0.1)
            except NoPackets:
                pass
            except Shutdown:
                break
            if netdata is not None:
                handle_network_data(netdata)


def main(net):
    stack = ProtocolStack(net)
    stack.main_loop()
    net.shutdown()
```

Note: Although the protocol stack example above uses a single Python thread to service both the from-network and from-application queues, it is possible to use multiple Python threads. The socket emulation library (discussed next) is threadsafe, as is the library code that handles sending/receiving packets on network ports.

6.2.2 Switchyard's socket emulation library

Switchyard provides a module similar to Python's built-in `socket` module that contains clones of many of the methods, functions and other items in the built-in module. We refer to the Switchyard socket module as an *emulation* module since it emulates the semantics of methods in the built-in module. The only line of code required to take advantage of Switchyard's socket emulation module is the import line. Instead of using importing a module named `socket`, you must import a module named `switchyard.lib.socket`. The `from ... import *` idiom is generally discouraged in Python, and a way to avoid this while isolating the change in a socket application to a single line is to do the following:

```
# instead of:
import socket

# to use Switchyard's socket emulation module, do:
import switchyard.lib.socket as socket
```

When using the suggested modification above, any use of attributes within the socket module (either built-in or emulated) can just be prefixed with `socket.` as normal. Note that in the code below, bytes objects are sent and received using `sendto` and `recvfrom`. (This same code is available in the `examples` folder in the Switchyard github repo.)

```
#!/usr/bin/env python3

# import socket
import switchyard.lib.socket as socket

HOST = '127.0.0.1'
PORT = 10000
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.settimeout(2.0)

print("Sending message to server at {},{}".format(HOST,PORT))
s.sendto(b'Hello, stack', (HOST,PORT))
try:
    data,addr = s.recvfrom(1024)
    print('Client socket application received message from {}: {}'.format(repr(addr),data.decode(
↪'utf8')))
except:
    print("Timeout")

s.close()
```

There are some key limitations and other issues to be aware of with Switchyard's socket emulation component:

- The most important limitation is that **only UDP sockets are supported**. Attempting to create any other type of socket will result in failure. Other socket types and support for using arbitrary protocol numbers may be supported in the future. As a result, there are a few socket object method calls that are not supported, such as `listen` and `accept`.
- The `create_connection` and `socketpair` calls are not available.
- The `getsockopt` and `setsockopt` calls are not currently supported, but may be in a future version.
- The various DNS-related calls in the socket module (e.g., `gethostbyname`, etc.) are available and simply handed off to the built-in `socket` module. Switchyard does not implement any DNS capability directly. Same for the byte-ordering calls (e.g., `ntohs`, `ntohl`, etc.)
- Switchyard attempts to be careful about choosing a local (ephemeral) port number for use, but its approach isn't fool-proof. There may be problems that arise due to a host OS using a local port that was already being used by Switchyard, but these situations should be rare in occurrence.

Note: Switchyard implements the socket layer by attempting to mirror, as closely as possible, the same constants, classes, and functions in the built-in socket module. It maintains a shared (threadsafe) queue that handles all data passed *down* from a socket application, and creates a separate queue for each socket for handling data being passed *up* to an application. As a result, Switchyard can support an application using *multiple* sockets at the same time (as long as they're all UDP!).

6.2.3 Starting socket applications with `swyard`

There is one additional command-line option for `swyard` when using a socket emulation application. The `-a` is used to specify the name of a file that contains the application-layer socket program.

The `-a` option can be used in conjunction with a Switchyard test scenario. If you want to test that a socket application emits a packet, then receives a packet from some "remote" host, you could create an expectation that a packet is emitted and an expectation that some other packet is received. You may need to use the

`copyfromlastout` argument when creating the `PacketInputEvent`, since the test scenario may not actually know what local port is being used by an application (among other things).

For example, to run a particular test scenario as well as an application program, the command line might look like the following:

```
$ swyard -a clientapp_udpstackex.py -t udpstack_tests.py udpstack.py
```

Note that the Python files used in the command line above are available in the `examples` folder of the Switchyard github repo.

To run in *live* mode, simply remove the `-t` option. Note that there is a server program in the `examples` folder that can be run with this code in live mode: you can see that the Switchyard-based UDP stack and associated client-side program can interact correctly with a "regular" Python UDP-based server program.

One final limitation to be aware of: only one socket application can be started by Switchyard at a time. This limitation may change in a future version.

Finally, note that Switchyard currently does not have any capabilities for testing the behavior of an application-layer socket program. The application code could use calls to `assert()` to verify that certain things happen as expected within the application, but there are no specific Switchyard features to help with this.

Note: When using Switchyard to create a protocol stack and run a socket-based application on a standard commodity operating system (e.g., a desktop/laptop Linux or macOS system), you may need to be careful about configuring the host firewall settings when starting Switchyard in real/live mode. In particular, any packets that you want Switchyard to handle should be explicitly *blocked* from the host operating system (or the host OS may respond in addition to Switchyard responding). It may also be helpful to explicitly bind your application socket to a particular port in order to limit the number of protocols and/or ports that need to be blocked from the host OS.

Note that when Switchyard is started with the `-a` flag and is thus starting an application-layer socket program, its default behavior with respect to the firewall is different. Normally, Switchyard blocks the host OS from receiving *any* traffic, but when executing an application-layer program *no* traffic is blocked, by default.

Refer to the section on *Firewall options* for command-line options to `swyard` to ensure that you block the correct traffic.

CHAPTER

SEVEN

INSTALLING SWITCHYARD

Switchyard has been tested and developed on the following operating systems:

- macOS 10.10 and later
- Ubuntu LTS releases from 14.04 and later
- Fedora 21

Note that these are all Unix-based systems. Switchyard may be enhanced in the future to support Windows-based systems. Ubuntu (current LTS) and macOS receive the most testing of Unix-based operating systems.

—

The steps for getting Switchyard up and running are as follows:

0. Install Python 3.4 or later, if you don't already have it.
1. Install any necessary libraries for your operating system.
2. Create an Python "virtual environment" for installing Python modules (or install the modules to your system Python)
3. Install Switchyard.

For step 0, you're on your own. Go to https://www.python.org/downloads/, or install packages via your OS'es package system, or use homebrew if you're on a Mac. Have fun.

The specific libraries necessary for different OSes (step 1) are described below, but steps 2 and 3 are the same for all operating systems and are covered next.

The recommended install procedure is to create a Python virtual environment for installing Switchyard and other required Python modules. One way to create a new virtual environment is to execute the following at a command line (in the folder in which you want to create the virtual environment):

```
$ python3 -m venv syenv
```

This command will create a new virtual environment called `syenv`. Once that's done, you can "activate" that environment and install Switchyard as follows:

```
$ source ./syenv/bin/activate
(syenv)$ python3 -m pip install switchyard
```

That's it. Once you've done that, the `swyard` program should be on your `PATH` (you can check by typing `which swyard`). If you no longer want to use the Python virtual environment you've created, you can just type `deactivate`.

7.1 Operating system-specific instructions

7.1.1 MacOS X

The easiest way to get Switchyard running in macOS is to install homebrew. You can use `brew` to install Python 3. You should also `brew` to install the `libpcap` package. That should be all that is necessary.

7.1.2 Ubuntu

For Ubuntu systems, you'll need to use `apt-get` or something similar to install the following packages:

```
libffi-dev libpcap-dev python3-dev python3-pip python3-venv
```

7.1.3 Fedora/RedHat

For Fedora and RedHat-based systems, you'll need to use `yum` or something similar to install a similar set of packages as with Ubuntu (but with the right name changes for the way packages are identified on Fedora):

```
libffi-devel libpcap-devel python3-devel python3-pip python3-virtualenv
```

CHAPTER

EIGHT

API REFERENCE

Before getting into all the details, it is important to note that all the below API features can be imported through the module `switchyard.lib.userlib`. This is a wrapper module to facilitate easy import of the various modules, functions, classes, and other items needed from the perspective of a user program in Switchyard.

Unless you are concerned about namespace pollution, importing all Switchyard symbols into your program can be done with the following:

```
from switchyard.lib.userlib import *
```

8.1 Net object reference

The *net* object is used for sending and receiving packets on network interfaces/ports. The API documentation below is for a base class that defines the various methods on a net object; there are two classes that derive from this base class which help to implement Switchyard's test mode and Switchyard's live network mode.

class `switchyard.llnetbase.`**LLNetBase**(*name=None*)

Base class for the low-level networking library in Python. "net" objects are constructed from classes derived from this class.

An object of this class is passed into the main function of a user's Switchyard program. Using methods on this object, a user can send/receive packets and query the device for what interfaces are available and how they are configured.

interface_by_ipaddr(*ipaddr*)

Given an IP address, return the interface that 'owns' this address

interface_by_macaddr(*macaddr*)

Given a MAC address, return the interface that 'owns' this address

interface_by_name(*name*)

Given a device name, return the corresponding interface object

interfaces()

Return a list of interfaces incident on this node/router. Each item in the list is an Interface object, each of which includes name, ethaddr, ipaddr, and netmask attributes.

port_by_ipaddr(*ipaddr*)

Alias for interface_by_ipaddr

port_by_macaddr(*macaddr*)

Alias for interface_by_macaddr

port_by_name(*name*)
Alias for interface_by_name

ports()
Alias for interfaces() method.

recv_packet(*timeout=None*)
Receive a packet on any port/interface. If a non-None timeout is given, the method will block for up to timeout seconds. If no packet is available, the exception NoPackets will be raised. If the Switchyard framework is being shut down, the Shutdown exception will be raised. If a packet is available, the ReceivedPacket named tuple (timestamp, input_port, packet) will be returned.

send_packet(*output_port*, *packet*)
Send a packet out the given output port/interface. Returns None.

testmode
Returns True if running in test mode and False if running in live/real mode.

8.2 Interface and InterfaceType reference

The `InterfaceType` enumeration is referred to by the `Interface` class, which encapsulates information about a network interface/port. The `InterfaceType` defines some basic options for types of interfaces:

class `switchyard.lib.interface.`**InterfaceType**
An enumeration.

Unknown=1

Loopback=2

Wired=3

Wireless=4

The `Interface` class is used to encapsulate information about a network interface:

class `switchyard.lib.interface.`**Interface**(*name*, *ethaddr*, *ipaddr=None*, *netmask=None*, *ifnum=None*, *iftype=<InterfaceType.Unknown: 1>*)

ethaddr
Get the Ethernet address associated with the interface

ifnum
Get the interface number (integer) associated with the interface

iftype
Get the type of the interface as a value from the InterfaceType enumeration.

ipaddr
Get the IPv4 address associated with the interface

ipinterface
Returns the address assigned to this interface as an IPInterface object. (see documentation for the built-in ipaddress module).

name
Get the name of the interface

netmask
Get the IPv4 subnet mask associated with the interface

8.3 Ethernet and IP addresses

Switchyard uses the built-in `ipaddress` module to the extent possible. Refer to the Python library documentation for details on the `IPv4Address` class and related classes. As noted in the source code, the `EthAddr` class based on source code from the POX Openflow controller.

class switchyard.lib.address.**EthAddr**(*addr=None*)
An Ethernet (MAC) address type.

isBridgeFiltered()
Checks if address is an IEEE 802.1D MAC Bridge Filtered MAC Group Address

This range is 01-80-C2-00-00-00 to 01-80-C2-00-00-0F. MAC frames that have a destination MAC address within this range are not relayed by bridges conforming to IEEE 802.1D

isGlobal()
Returns True if this is a globally unique (OUI enforced) address.

isLocal()
Returns True if this is a locally-administered (non-global) address.

isMulticast()
Returns True if this is a multicast address.

is_bridge_filtered

is_global

is_local

is_multicast

packed

raw
Returns the address as a 6-long bytes object.

toRaw()

toStr(*separator=':'*)
Returns the address as string consisting of 12 hex chars separated by separator.

toTuple()
Returns a 6-entry long tuple where each entry is the numeric value of the corresponding byte of the address.

There are two enumeration classes that hold special values for the IPv4 and IPv6 address families. Note that since these classes derive from `enum`, you must use `name` to access the name attribute and `value` to access the value (address) attribute.

class switchyard.lib.address.**SpecialIPv4Addr**
An enumeration.

IP_ANY = ip_address("0.0.0.0")

IP_BROADCAST = ip_address("255.255.255.255")

class switchyard.lib.address.**SpecialIPv6Addr**
An enumeration.

UNDEFINED = ip_address('::')

ALL_NODES_LINK_LOCAL = ip_address('ff02::1')

ALL_ROUTERS_LINK_LOCAL = ip_address('ff02::2')

```
ALL_NODES_INTERFACE_LOCAL = ip_address('ff01::1')
ALL_ROUTERS_INTERFACE_LOCAL = ip_address('ff01::2')
```

8.4 Packet parsing and construction reference

class switchyard.lib.packet.**Packet**(*raw=None*, *first_header=None*)
Base class for packet headers.

The Packet class acts as a container for packet headers. The + and += operators are defined for use with the Packet class to add on headers (to the end of the packet). Indexing can also be done with Packet objects to access individual header objects. Indexes may be integers (from 0 up to, but not including, the number of packet headers), or indexes may also be packet header class names. Exceptions are raised for invaliding indexing of either kind.

The optional raw parameter can accept a bytes object, which assumed to be a serialized packet to be reconstructed. The optional parameter first_header indicates the first header of the packet to be reconstructed, which defaults to Ethernet.

```
>>> p = Packet()
>>> p += Ethernet()
>>> p[0]
<switchyard.lib.packet.ethernet.Ethernet object at 0x10632bb08>
>>> p[Ethernet]
<switchyard.lib.packet.ethernet.Ethernet object at 0x10632bb08>
>>> str(p)
'Ethernet 00:00:00:00:00:00->00:00:00:00:00:00 IP'
>>> str(p[0])
'Ethernet 00:00:00:00:00:00->00:00:00:00:00:00 IP'
>>> str(p[Ethernet])
'Ethernet 00:00:00:00:00:00->00:00:00:00:00:00 IP'
>>>
```

add_header(*ph*)
Add a PacketHeaderBase derived class object, or a raw bytes object as the next "header" item in this packet. Note that 'header' may be a slight misnomer since the last portion of a packet is considered application payload and not a header per se.

add_payload(*ph*)
Alias for add_header

static from_bytes(*raw*, *first_header*)
Create a new packet by parsing the contents of a bytestring

get_header(*hdrclass*, *returnval=None*)
Return the first header object that is of class hdrclass, or None if the header class isn't found.

get_header_by_name(*hdrname*)
Return the header object that has the given (string) header class name. Returns None if no such header exists.

get_header_index(*hdrclass*, *startidx=0*)
Return the first index of the header class hdrclass starting at startidx (default=0), or -1 if the header class isn't found in the list of headers.

has_header(*hdrclass*)
Return True if the packet has a header of the given hdrclass, False otherwise.

headers()
Return a list of packet header names in this packet.

insert_header(*idx*, *ph*)
Insert a PacketHeaderBase-derived object at index idx the list of headers. Any headers previously in the Packet from index idx:len(ph) are shifted to make room for the new packet.

num_headers()
Return the number of headers in the packet.

prepend_header(*ph*)
Insert a PacketHeader object at the beginning of this packet (i.e., as the first header of the packet).

size()
Return the packed length of this header

to_bytes()
Returns serialized bytes object representing all headers/ payloads in this packet

To delete/remove a header, you can use the `del` operator as if the packet object is a Python list:

```
>>> del p[0] # delete/remove first header in packet
>>>
```

You can assign new header objects to a packet by integer index, but not by packet header class index:

```
>>> p[0] = Ethernet() # assign a new Ethernet header to index 0
>>>
```

8.4.1 Header classes

In this section, detailed documentation for all packet header classes is given. For each header class, there are three common *instance* methods that may be useful and which are *not* documented below for clarity. They are defined in the base class `PacketHeaderBase`. Note that any new packet header classes that derive from `PacketHeaderBase` must implement these three methods.

class switchyard.lib.packet.**PacketHeaderBase**(***kwargs*)
Base class for packet headers.

from_bytes(*raw*)
Reconstruct the attributes of a header given the bytes object named raw. The method returns any bytes that are *not* used to reconstruct a header. An exception (typically a ValueError) is raised if there is some kind of problem deserializing the bytes object into packet header attributes.

size()
Returns the number of bytes that the header would consist of when serialized to wire format

to_bytes()
Return a 'packed' byte-level representation of this packet header.

There are also three common *class* methods that are used when creating a new packet header class (see *Creating new packet header types*).

class switchyard.lib.packet.**PacketHeaderBase**(***kwargs*)
Base class for packet headers.

classmethod add_next_header_class(*attr*, *hdrcls*)
Add a new mapping between a next header type value and a Python class that implements that header type.

classmethod set_next_header_class_key(*attr*)
Indicate which attribute is used to decide the type of packet header that comes after this one. For example, the IPv4 protocol attribute.

classmethod set_next_header_map(*mapdict*)
(Re)initialize a dictionary that maps a "next header type" attribute to a Python class that implements that header type.

8.4.2 Ethernet header

class switchyard.lib.packet.**Ethernet**(***kwargs*)
Represents an Ethernet header with fields src (source Ethernet address), dst (destination Ethernet address), and ethertype (type of header to come in the packet after the Ethernet header). All valid ethertypes are defined below.

dst

ethertype

src

class switchyard.lib.packet.common.**EtherType**
An enumeration.

IP = 0x0800

IPv4 = 0x0800

ARP = 0x0806

x8021Q = 0x8100

IPv6 = 0x86dd

SLOW = 0x8809

MPLS = 0x8847

x8021AD = 0x88a8

LLDP = 0x88cc

x8021AH = 0x88e7

IEEE8023 = 0x05dc

The EtherType class is derived from the built-in Python Enumerated class type. Note that some values start with 'x' since they must start with an alphabetic character to be valid in the enum.

By default, the Ethernet header addresses are all zeroes ("00:00:00:00:00:00"), and the ethertype is IPv4. Here is an example of creating an Ethernet header and setting the header fields to non-default values:

```
>>> e = Ethernet()
>>> e.src = "de:ad:00:00:be:ef"
>>> e.dst = "ff:ff:ff:ff:ff:ff"
>>> e.ethertype = EtherType.ARP
```

As with all packet header classes, keyword parameters can be used to initialize header attributes:

```
>>> e = Ethernet(src="de:ad:00:00:be:ef", dst="ff:ff:ff:ff:ff:ff", ethertype=EtherType.ARP)
```

8.4.3 ARP (address resolution protocol) header

class switchyard.lib.packet.**Arp**(***kwargs*)

hardwaretype

operation

protocoltype

senderhwaddr

senderprotoaddr

targethwaddr

targetprotoaddr

class switchyard.lib.packet.common.**ArpOperation**
An enumeration.

Request = 1

Reply = 2

The `Arp` class is used for constructing ARP (address resolution protocol) requests and replies. The `hardwaretype` property defaults to `Ethernet`, so you don't need to set that when an `Arp` object is instantiated. The operation can be set using the enumerated type `ArpOperation`, as indicated above. The remaining fields hold either `EthAddr` or `IPv4Address` objects, and can be initialized using string representations of Ethernet or IPv4 addresses as appropriate. Below is an example of creating an ARP request. You can assume in the example that the senders Ethernet and IPv4 addresses are `srchw` and `srcip`, respectively. You can also assume that the IPv4 address for which we are requesting the Ethernet address is `targetip`.

```
ether = Ethernet()
ether.src = srchw
ether.dst = 'ff:ff:ff:ff:ff:ff'
ether.ethertype = EtherType.ARP
arp = Arp(operation=ArpOperation.Request,
          senderhwaddr=srchw,
          senderprotoaddr=srcip,
          targethwaddr='ff:ff:ff:ff:ff:ff',
          targetprotoaddr=targetip)
arppacket = ether + arp
```

8.4.4 IP version 4 header

class switchyard.lib.packet.**IPv4**(***kwargs*)
Represents an IP version 4 packet header. All properties relate to specific fields in the header and can be inspected and/or modified.

Note that the field named "hl" ("h-ell") stands for "header length". It is the size of the header in 4-octet quantities. It is a read-only property (cannot be set).

Note also that some IPv4 header option classes are available in Switchyard, but are currently undocumented.

dscp

dst

ecn

flags

fragment_offset

hl

ipid

options

protocol

src

tos

total_length

ttl

class switchyard.lib.packet.common.**IPProtocol**
An enumeration.

ICMP = 1

TCP = 6

UDP = 17

The IPProtocol class derives from the Python 3-builtin Enumerated class type. There are other protocol numbers defined. See switchyard.lib.packet.common for all defined values.

A just-constructed IPv4 header defaults to having all zeroes for the source and destination addresses ('0.0.0.0') and the protocol number defaults to ICMP. An example of creating an IPv4 header and setting various fields is shown below:

```
>>> ip = IPv4()
>>> ip.srcip = '10.0.1.1'
>>> ip.dstip = '10.0.2.42'
>>> ip.protocol = IPProtocol.UDP
>>> ip.ttl = 64
```

8.4.5 UDP (user datagram protocol) header

class switchyard.lib.packet.**UDP**(***kwargs*)
The UDP header contains just source and destination port fields.

dst

src

To construct a packet that includes an UDP header as well as some application data, the same pattern of packet construction can be followed:

```
>>> p = Ethernet() + IPv4(protocol=IPProtocol.UDP) + UDP()
>>> p[UDP].src = 4444
>>> p[UDP].dst = 5555
>>> p += b'These are some application data bytes'
>>> print (p)
```

```
Ethernet 00:00:00:00:00:00->00:00:00:00:00:00 IP | IPv4 0.0.0.0->0.0.0.0 UDP | UDP 4444->5555 |␣
↪RawPacketContents (37 bytes) b'These are '...
>>>
```

Note that we didn't set the IP addresses or Ethernet addresses above, but did set the IP protocol to correctly match the next header (UDP). Adding a payload to a packet is as simple as tacking on a Python `bytes` object. You can also construct a `RawPacketContents` header, which is just a packet header class that wraps a set of raw bytes.

8.4.6 TCP (transmission control protocol) header

class switchyard.lib.packet.**TCP**(***kwargs*)

Represents a TCP header. Includes properties to access/modify TCP header fields.

ACK

CWR

ECE

FIN

NS

PSH

RST

SYN

URG

ack

dst

flags

flagstr

offset

options

seq

src

urgent_pointer

window

Setting TCP header flags can be done by assigning 1 to any of the mnemonic flag properties:

```
>>> t = TCP()
>>> t.SYN = 1
```

To check whether a flag has been set, you can simply inspect the the flag value:

```
>>> if t.SYN:
>>> ...
```

8.4.7 ICMP (Internet control message protocol) header

class switchyard.lib.packet.**ICMP**(***kwargs*)

A mother class for all ICMP message types. It holds a reference to another object that contains the specific ICMP data (icmpdata), given a particular ICMP type. Just setting the icmptype causes the data object to change (the change happens automatically when you set the icmptype). The icmpcode field will also change, but it only changes to some valid code given the new icmptype.

Represents an ICMP packet header.

icmpcode

icmpdata

icmptype

class switchyard.lib.packet.common.**ICMPType**

An enumeration.

EchoReply = 0

DestinationUnreachable = 3

SourceQuench = 4

Redirect = 5

EchoRequest = 8

TimeExceeded = 11

The icmptype and icmpcode header fields determine the value stored in the icmpdata property. When the icmptype is set to a new value, the icmpdata field is *automatically* set to the correct object.

```
>>> i = ICMP()
>>> print (i)
ICMP EchoRequest 0 0 (0 data bytes)
>>> i.icmptype = ICMPType.TimeExceeded
>>> print (i)
ICMP TimeExceeded:TTLExpired 0 bytes of raw payload (b'') OrigDgramLen: 0
>>> i.icmpcode
<ICMPCodeTimeExceeded.TTLExpired: 0>
>>> i.icmpdata
<switchyard.lib.packet.icmp.ICMPTimeExceeded object at 0x10d3a3308>
```

Notice above that when the icmptype changes, other contents in the ICMP header object change appropriately.

To access and/or modify the *payload* (i.e., data) that comes after the ICMP header, use `icmpdata.data`. This object is a raw bytes object and can be accessed and or set. For example, with many ICMP error messages, up to the first 28 bytes of the "dead" packet should be included, starting with the IPv4 header. To do that, you must set the `icmpdata.data` attribute with the byte-level representation of the IP header data you want to include, as follows:

```
>>> i.icmpdata.data
b''
>>> i.icmpdata.data = pkt.to_bytes()[:28]
>>> i.icmpdata.origdgramlen = len(pkt)
>>> print (i)
ICMP TimeExceeded:TTLExpired 28 bytes of raw payload (b'E\x00\x00\x14\x00\x00\x00\x00\x00\x01')␣
↪OrigDgramLen: 42
>>>
```

In the above code segment, `pkt` should be a Packet object that just contains the IPv4 header and any subsequent headers and data. It must *not* include an Ethernet header. If you need to strip an Ethernet header, you can get its index (`pkt.get_header_index(Ethernet)`), then remove the header by index (`del pkt[index]`).

Notice that above, the `to_bytes` method returns the byte-level representation of the IP header we're including as the payload. The `to_bytes` method can be called on any packet header, or on an packet object (in which case *all* packet headers will be byte-serialized).

To set the icmpcode, a dictionary called `ICMPTypeCodeMap` is defined in `switchyard.lib.packet`. Keys in the dictionary are of type `ICMPType`, and values for each key is another enumerated type indicating the valid codes for the given type.

```
>>> from switchyard.lib.packet import *
>>> ICMPTypeCodeMap[ICMPType.DestinationUnreachable]
<enum 'DestinationUnreachable'>
```

Just getting the dictionary value isn't particularly helpful, but if you coerce the enum to a list, you can see all valid values:

```
>>> list(ICMPTypeCodeMap[ICMPType.DestinationUnreachable])
[ <DestinationUnreachable.ProtocolUnreachable: 2>,
  <DestinationUnreachable.SourceHostIsolated: 8>,
  <DestinationUnreachable.FragmentationRequiredDFSet: 4>,
  <DestinationUnreachable.HostUnreachable: 1>,
  <DestinationUnreachable.DestinationNetworkUnknown: 6>,
  <DestinationUnreachable.NetworkUnreachableForTOS: 11>,
  <DestinationUnreachable.HostAdministrativelyProhibited: 10>,
  <DestinationUnreachable.DestinationHostUnknown: 7>,
  <DestinationUnreachable.HostPrecedenceViolation: 14>,
  <DestinationUnreachable.PrecedenceCutoffInEffect: 15>,
  <DestinationUnreachable.NetworkAdministrativelyProhibited: 9>,
  <DestinationUnreachable.NetworkUnreachable: 0>,
  <DestinationUnreachable.SourceRouteFailed: 5>,
  <DestinationUnreachable.PortUnreachable: 3>,
  <DestinationUnreachable.CommunicationAdministrativelyProhibited: 13>,
  <DestinationUnreachable.HostUnreachableForTOS: 12> ]
```

Another example, but with the much simpler EchoRequest:

```
>>> list(ICMPTypeCodeMap[ICMPType.EchoRequest])
[<EchoRequest.EchoRequest: 0>]
```

If you try to set the icmpcode to an invalid value, an exception will be raised:

```
>>> i = ICMP()
>>> i.icmptype = ICMPType.DestinationUnreachable
>>> i.icmpcode = 44
Traceback (most recent call last):
...
>>>
```

You can either (validly) set the code using an integer, or a valid enumerated type value:

```
>>> i.icmpcode = 2
>>> print(i)
ICMP DestinationUnreachable:ProtocolUnreachable 0 bytes of raw payload (b'') NextHopMTU: 0
>>> i.icmpcode = ICMPTypeCodeMap[i.icmptype].HostUnreachable
```

```
>>> print (i)
ICMP DestinationUnreachable:HostUnreachable 0 bytes of raw payload (b'') NextHopMTU: 0
```

Below are shown the ICMP data classes, as well as any properties that can be inspected and/or modified on them.

class switchyard.lib.packet.**ICMPEchoReply**

data

identifier

sequence

class switchyard.lib.packet.**ICMPDestinationUnreachable**

data

nexthopmtu

origdgramlen

class switchyard.lib.packet.**ICMPSourceQuench**

data

class switchyard.lib.packet.**ICMPRedirect**

data

redirectto

class switchyard.lib.packet.**ICMPEchoRequest**

data

identifier

sequence

class switchyard.lib.packet.**ICMPTimeExceeded**

data

origdgramlen

8.5 Test scenario creation

class switchyard.lib.testing.**TestScenario**(*name*)
Test scenario definition. Given a list of packetio event objects, generates input events and tests/verifies output events.

add_file(*fname*, *text*)

add_interface(*interface_name*, *macaddr*, *ipaddr=None*, *netmask=None*, ***kwargs*)
Add an interface to the test scenario.

(str, str/EthAddr, str/IPAddr, str/IPAddr) -> None

expect(*event*, *description*)
Add a new event and description to the expected set of events to occur for this test scenario.

(Event object, str) -> None

interfaces()

name

ports()
Alias for interfaces() method.

class switchyard.lib.testing.**PacketInputEvent**(*device*, *packet*, *display=None*, *copyfromlastout=None*)
Test event that models a packet arriving at a router/switch (e.g., a packet that we generate).

match(*evtype*, ***kwargs*)
Does event type match me? PacketInputEvent currently ignores any additional arguments.

class switchyard.lib.testing.**PacketInputTimeoutEvent**(*timeout*)
Test event that models a timeout when trying to receive a packet. No packet arrives, so the switchy app should handle a NoPackets exception and continue

match(*evtype*, ***kwargs*)
Does event type match me? PacketInputEvent currently ignores any additional arguments.

class switchyard.lib.testing.**PacketOutputEvent**(**args*, ***kwargs*)
Test event that models a packet that should be emitted by a router/switch.

match(*evtype*, ***kwargs*)
Does event type match me? PacketOutputEvent requires two additional keyword args: device (str) and packet (packet object).

8.6 Application-layer

Two static methods on the ApplicationLayer class are used to send messages up a socket application and to receive messages from socket applications.

class switchyard.lib.socket.**ApplicationLayer**

static recv_from_app(*timeout=None*)
Called by a network stack implementer to receive application-layer data for sending on to a remote location.

Can optionally take a timeout value. If no data are available, raises NoPackets exception.

Returns a 2-tuple: flowaddr and data. The flowaddr consists of 5 items: protocol, localaddr, localport, remoteaddr, remoteport.

static send_to_app(*proto*, *local_addr*, *remote_addr*, *data*)
Called by a network stack implementer to push application-layer data "up" from the stack.

Arguments are protocol number, local_addr (a 2-tuple of IP address and port), remote_addr (a 2-tuple of IP address and port), and the message.

Returns True if a socket was found to which to deliver the message, and False otherwise. When False is returned, a log warning is also emitted.

Switchyard's socket emulation module is intended to follow, relatively closely, the methods and attributes available in the built-in `socket` module.

class `switchyard.lib.socket.`**socket**(*family*, *socktype*, *proto=0*, *fileno=0*)
A socket object, emulated by Switchyard.

accept()
Not implemented.

bind(*address*)
Alter the local address with which this socket is associated. The address parameter is a 2-tuple consisting of an IP address and port number.

NB: this method fails and returns -1 if the requested port to bind to is already in use but does *not* check that the address is valid.

close()
Close the socket.

connect(*address*)
Set the remote address (IP address and port) with which this socket is used to communicate.

connect_ex(*address*)
Set the remote address (IP address and port) with which this socket is used to communicate.

family
Get the address family of the socket.

getpeername()
Return a 2-tuple containing the remote IP address and port associated with the socket, if any.

getsockname()
Return a 2-tuple containing the local IP address and port associated with the socket.

getsockopt(*level*, *option*, *buffersize=0*)
Not implemented.

gettimeout()
Obtain the currently set timeout value.

listen(*backlog*)
Not implemented.

proto
Get the protocol of the socket.

recv(*buffersize*, *flags=0*)
Receive data on the socket. The buffersize and flags arguments are currently ignored. Only returns the data.

recv_into(**args*)
Not implemented.

recvfrom(*buffersize*, *flags=0*)
Receive data on the socket. The buffersize and flags arguments are currently ignored. Returns the data and an address tuple (IP address and port) of the remote host.

recvfrom_into(**args*)
Not implemented.

recvmsg(**args*)
Not implemented.

send(*data*, *flags=0*)
Send data on the socket. A call to connect() must have been previously made for this call to succeed. Flags is currently ignored.

sendall(**args*)
Not implemented.

sendmsg(**args*)
Not implemented.

sendto(*data*, **args*)
Send data on the socket. Accepts the same parameters as the built-in socket sendto: data[, flags], address where address is a 2-tuple of IP address and port. Any flags are currently ignored.

setblocking(*flags*)
Set whether this socket should block on a call to recv*.

setsockopt(**args*)
Not implemented.

settimeout(*timeout*)
Set the timeout value for this socket.

shutdown(*flag*)
Shut down the socket. This is currently implemented by calling close().

timeout
Obtain the currently set timeout value.

type
Get the type of the socket.

8.7 Utility functions

switchyard.lib.logging.**log_failure**(*s*)
Convenience function for failure message.

switchyard.lib.logging.**log_warn**(*s*)
Convenience function for warning message.

switchyard.lib.logging.**log_info**(*s*)
Convenience function for info message.

switchyard.lib.logging.**log_debug**(*s*)
Convenience function for debugging message.

switchyard.lib.debugging.**debugger**()
Invoke the interactive debugger. Can be used anywhere within a Switchyard program.

CHAPTER

NINE

RELEASE NOTES

The headings below refer either to branches on Switchyard's github repo (v1 and v2) or tags (2017.01.1).

9.1 2017.01.2

Add the capability to pass arguments to a Switchyard program via `-g` option to `swyard`. Switchyard parses and assembles `*args` and `**kwargs` to pass into the user code, being careful to only pass them if the code can accept them.

9.2 2017.01.1

Major revision; expansion of types of exercises supported (notably application-layer programs via socket emulation) and several non-backward compatible API changes. Simplified user code import (single import of switchyard.lib.userlib). Installation via standard setuptools, so easily installed via easy_install or pip. Major revision of documentation. Lots of new tests were written, bringing test coverage above 90%. Expansion of exercises is still in progress.

Some key API changes to be aware of:

- the Scenario class is renamed TestScenario. The PacketOutputEvent previously allowed Openflow 1.0-like wildcard strings to specify wildcards for matching packets; these strings are no longer supported. To specify wildcards, a tuple of (classname,attribute) must be used; refer to *Test scenario creation*, above.
- `recv_packet` *always* returns a timestamp now; it returns a 3-tuple (named tuple) of timestamp, input_port and packet.
- The only import required by user code is switchyard.lib.userlib, although individual imports are still fine (just more verbose).
- Instead of invoking `srpy.py`, a `swyard` program is installed during the new install process. `swyard` has a few command-line changes compared with `srpy.py`. In particular, the `-s` option has gone away; to run Switchyard with a test, just use the `-t` option with the scenario file as the argument.

9.3 v2

Complete rewrite of v1. Moved to Python 3 and created packet parsing libraries, new libpcap interface library (pcapffi). Redesigned test scenario modules and an expanded of publicly available exercises. Used at Colgate twice and University of Wisconsin-Madison twice. Available on the `v2` branch on github.

9.4 v1

First version, which used the POX packet parsing libraries and had a variety of limitations. Implemented in Python 2 and used at Colgate once. Available on the `v1` branch on github, but very much obsolete.

CHAPTER

TEN

ACKNOWLEDGMENTS AND THANKS

Once again, I gratefully acknowledge support from the NSF. The materials here are based upon work supported by the National Science Foundation under grant CNS-1054985 ("CAREER: Expanding the functionality of Internet routers"). 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.

Thanks to Colgate COSC465 students from Spring 2014 and Spring 2015 for being guinea pigs and giving feedback for the very first versions of Switchyard. Thanks also to Prof. Paul Barford and CS640 students at the University of Wisconsin for using and providing feedback on Switchyard.

Thanks to those students who have contributed fixes and made suggestions for improvements. In particular:

- Thanks to Saul Shanabrook for several specific suggestions and bug reports that have led to improvements in Switchyard.
- Thanks to Xuyi Ruan for identifying and suggesting a fix to bugs on one of the documentation diagrams.
- Thanks to Sean Wilson for a bug fix on an infinitely recursive property setter. Oops, but this dumb bug motivated me to significantly improve test coverage, so there's that.
- Thanks to Leon Yang for identifying a problem with kwarg processing for ICMP.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

I gratefully acknowledge support from the NSF. The materials here are based upon work supported by the National Science Foundation under grant CNS-1054985 (“CAREER: Expanding the functionality of Internet routers”).

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.

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W