

Packet Construction Set

George V. Neville-Neil

July 13, 2006

1 Introduction

PCS is a set of Python modules and objects that make building network protocol testing tools easier for the protocol developer. The core of the system is the `pcs` module itself which provides the necessary functionality to create classes that implement packets.

Installing PCS is covered in the text file, `INSTALLATION`, which came with this package. The code is under a BSD License and can be found in the file `COPYRIGHT` in the root of this package.

In the following document we set `CLASSES` `functions()` and `methods()` apart by setting them in different type. Methods and functions are also followed by parentheses, “()”, which classes are not.

2 A Quick Tour

For the impatient programmer this section is a 5 minute intro to using PCS. Even faster than this tour would be to read some of the test code in the `tests` sub-directory or the scripts in the `scripts` sub directory.

PCS is a set of functions to encode and decode network packets from various formats as well as a set of *classes* for the most commonly use network protocols. Each object derived from a packet has fields automatically built into it that represent the relevant sections of the packet.

Let’s grab a familiar packet to work with, the IPv4 packet. IPv4 packets show a few interesting features of PCS. Figure 2 shows the definition of an IPv4 packet header from [?] which specifies the IPv4 protocol.

In PCS every packet class contains fields which represent the fields of the packet exactly, including their bit widths. Figure2 shows a command line interaction with an IPv4 packet.

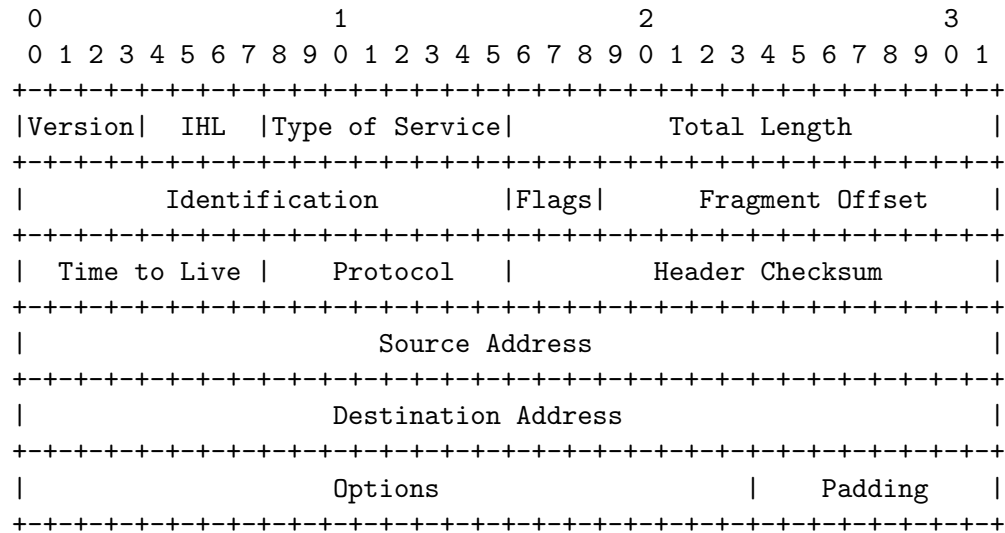


Figure 1: IPv4 Header Format

Each packet has a built in field called `bytes` which always contains the wire representation of the packet.

In Figure3 the `bytes` field has been changed in its first position by setting the `hlen` or header length field to 20, $5 \ll 2$. Such programmatic access is available to all fields of the packet.

The IPv4 header has fields that can be problematic to work with in any language including ones that are

fig:ipheadfeatures less than one byte (octet) in length (Version, IHL, Flags)

fig:ipheadfeatures not an even number of bits (Flags)

fig:ipheadfeatures not aligned on a byte boundary (Fragment Offset)

Using just these features it is possible to write complex programs in Python that directly manipulate packets. For now you should know enough to safely ignore this documentation until you to explore further.

3 Working with Packets

In PCS every packet is a class and the layout of the packet is defined by a Layout class which contains a set of Fields. Fields can be from 1 to many

```

Python 2.4.2 (#1, Mar  7 2006, 15:04:29)
[GCC 4.0.1 (Apple Computer, Inc. build 5250)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pcs.packets.ipv4 import *
>>> ip = ipv4()
>>> print ip
version 4
hlen 0
tos 0
length 0
id 0
flags 0
offset 0
ttl 64
protocol 0
checksum 0
src 0.0.0.0
dst 0.0.0.0

>>> ip.hlen=5<<2
>>> print ip
version 4
hlen 20
tos 0
length 0
id 0
flags 0
offset 0
ttl 64
protocol 0
checksum 0
src 0.0.0.0
dst 0.0.0.0

```

Figure 2: Quick and Dirty IPv4 Example

```

>>> from pcs.packets.ipv4 import *
>>> ip = ipv4()
>>> ip.bytes
'@\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> ip.hlen = 5 << 2
>>> ip.bytes
'D\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x00\x00\x00'

```

Figure 3: The bytes Field of the Packet

```

ip = ipv6()
assert (ip != None)
ip.traffic_class = 1
ip.flow = 0
ip.length = 64
ip.next_header = 6
ip.hop = 64
ip.src = inet_pton(AF_INET6, "::1")
ip.dst = inet_pton(AF_INET6, "::1")

```

Figure 4: IPv6 Class

bits, so it is possible to build packets with arbitrary width bit fields. Fields know about the widths and will throw exceptions when they are overloaded.

Every Packet object, that is an object instantiated from a specific PCS packet class, has a field named bytes which shows the representation of the data in the packet at that point in time. It is the bytes field that is used when transmitting the packet on the wire.

The whole point of writing PCS was to make it easier to experiment with various packet types. In PCS there are packet classes and packet objects. Packet classes define the named fields of the packet and these named fields are properties of the object. A practical example may help. Given an IPv6 packet class it is possible to create the object, set various fields, as well as transmit and receive the object.

A good example is the IPv6 class: The code in Figure 4 gets a new IPv6 object from the `ipv6()` class, which was imported earlier, and sets various fields in the packet. Showing the bytes field, Figure 5 gives us an idea of how well this is working.

Note that various bits are set throughout the bytes. The data in the

```
>>> ip.bytes
'\x10\x00\x00\x00@\x06@\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01'
```

Figure 5: Bytes of the IPv6 Packet

```
>>> print ip
version 6
traffic_class 1
flow 0
length 64
next_header 6
hop 64
src ::1
dst ::1
```

Figure 6: Printing a Packet

packet can be pretty printed using the `print()` function as seen in Figure 6 or it can be dumped as a string directly as seen in Figure 7.

```
>>> ip
<IPv6: src: 0, dst: 0, traffic_class: 0, flow: 0, length: 0, \
version:6, hop: 0, next_header: 0>
```

Figure 7: Using the `__repr__()` method

4 Creating Packet Classes

For a packet to be a part of PCS it must sub-classed from the `PACKET` class as seen in Figure ??

The code in Figure ?? defines a new class, one that will describe an IPv6 packet, sub-classed from the `PACKET` base class. All PCS packets must have a `layout` field, which is set via the base class's `__init__()` method, called at the end of the `ipv6` class's own `__init__()` method.

Part of packet initialization is to set up the fields that the packet will contain. Each field has several possible arguments, but the two that are

```

class ipv6(pcs.Packet):
    """A class that contains the IPv6 header. All other data is
    chained on the end."""

    layout = pcs.Layout()

    def __init__(self, bytes = None):
        """IPv6 Packet from RFC 2460"""
        version = pcs.Field("version", 4, default = 6)
        traffic = pcs.Field("traffic_class", 8)
        flow = pcs.Field("flow", 20)
        length = pcs.Field("length", 16)
        next = pcs.Field("next_header", 8)
        hop = pcs.Field("hop", 8)
        src = pcs.Field("src", 16 * 8, type = str)
        dst = pcs.Field("dst", 16 * 8, type = str)
        pcs.Packet.__init__(self,
                             [version, traffic, flow, length, next, hop,
                              src, dst], bytes)

```

Figure 8: IPv6 Packet Class

```
ip = ipv6()
ip.hop = 4 # Set hop count to 4.
```

Figure 9: Setting a Packet Field

required are a name, which is the string field specified as the first argument and a width in bits, specified as the second argument. Note that all field widths are specified in *bits* and not *bytes* or *octets*. Fields may also have a type (see below) and default values. The name of the field becomes a named property of the object which is what makes it possible to have code that like that in Figure 9 where we set the packet’s hop count via its `hop` property.

The fields are set by passing them as an array to the PCS base class initialization method.

5 Retrieving Packets

One of the uses of **PCS** is to analyze packets that have previously stored, for example by a program such as **tcpdump(1)**. **PCS** supports reading and writing **tcpdump(1)** files through the **pcap** library written by Doug Song. The python API exactly mirrors the C API in that packets are processed via a callback to a `dispatch()` routine, usually in a loop. Complete documentation on the **pcap** library can be found with its source code or on its web page. This document only explains **pcap** as it relates to how we use it in **PCS**.

When presented with a possibly unknown data file how can you start? If you don’t know the bottom layer protocol stored in the file, such as *Ethernet*, *FDDI*, or raw *IP* packets such as might be capture on a loopback interface, it’s going to be very hard to get your program to read the packets correctly. The **pcap** library handles this neatly for us. When opening a saved file it is possible to ask the file what kind of data it contains, through the `datalink()` method.

In Figure10 we see two different save files being opened. The first, `etherping.out` is a **tcpdump** file that contains data collected on an Ethernet interface, type `DLT_EN10` and the second, `looping.out` was collected from the *loopback* interface and so contains no Layer 2 packet information.

Not only do we need to know the type of the lowest layer packets but we also need to know the next layer’s offset so that we can find the end of the `datalink` packet and the beginning of the network packet. The `dloff` field

```

>>> import pcap
>>> efile = pcap.pcap("etherping.out")
>>> efile.datalink()
1
>>> efile.datalink() == pcap.DLT_EN10MB
True
>>> lfile = pcap.pcap("loopping.out")
>>> lfile.datalink()
0
>>> lfile.datalink() == pcap.DLT_NULL
True
>>> lfile.datalink() == pcap.DLT_EN10MB
False
>>>

```

Figure 10: Determining the Bottom Layer

```

>>> efile.dloff
14
>>> lfile.dloff
4
>>>

```

Figure 11: Finding the Datalink Offset

of the PCAP class gives the data link offset. Figure11 continues the example shown in Figure10 and shows that the Ethernet file has a datalink offset of 14 bytes, and the loopback file 4.

It is in the loopback case that the number is most important. Most network programmers remember that Ethernet headers are 14 bytes in length, but the 4 byte offset for loopback may seem confusing, and if forgotten any programs run on data collected on a loopback interface will appear as garbage.

With all this background we can now read a packet and examine it. Figure 12 shows what happens when we create a packet from a data file.

In this example we pre-suppose that the packet is an IPv4 packet but that is not actually necessary. We can start from the lowest layer, which in this case is Ethernet, because the capture file knows the link layer of the data. Packets are fully decoded as much as possible when they are read.


```

>>> ip = ipv4(packet[efile.dloff:len(packet)])
>>> print ip
version 4
hlen 5
tos 0
length 84
id 34963
flags 0
offset 0
ttl 64
protocol 1
checksum 58688
src 192.168.101.166
dst 169.229.60.161

```

Figure 12: Reading in a Packet

PCS is able to do this via a special method, called `next()` and a field called `data`. Every PCS class has a `next()` method which attempts to figure out the next higher layer protocol if there is any data in a packet beyond the header. If the packet's data can be understood and a higher layer packet class is found the `next()` creates a packet object of the appropriate type and sets the `data` field to point to the packet. This process is recursive, going up the protocol layers until all remaining packet data or higher layers are exhausted. In Figure 13 we see an example of an Ethernet packet which contains an IPv4 packet which contains an ICMPv4 packet all connected via their respective `data` fields.

6 Storing Packets

This section intentionally left blank.

Need to update **pcap** module to include support for true dump files.

7 Sending Packets

In **PCS** packets are received and transmitted (see 7 using **CONNECTORS**). A **CONNECTOR** is an abstraction that can contain a traditional network *socket*, or a file descriptor which points to a protocol filter such as *BPF*. For

```
>>> from pcs.packets.ethernet import ethernet
>>> ethernet = ethernet(packet[0:len(packet)])
>>> ethernet.data
<Packet: hlen: 5, protocol: 1, src: 3232261542L, tos: 0, dst: 2850372769L, ttl: 64, 1
>>> ip = ethernet.data
>>> print ethernet
src: 0:10:db:3a:3a:77
dst: 0:d:93:44:fa:62
type: 0x800
>>> print ip
version 4
hlen 5
tos 0
length 84
id 34963
flags 0
offset 0
ttl 64
protocol 1
checksum 58688
src 192.168.101.166
dst 169.229.60.161
```

Figure 13: Packet Decapsulation on Read

```

import pcs

from socket import *

def main():

    conn = pcs.TCP4Connector("127.0.0.1", 80)
    conn.write("GET / \n")
    result = conn.read(1024)

    print result

main()

```

Figure 14: HTTP Get Script

completely arbitrary reasons we will discuss packet transmission first.

In order to send a packet we must first have a connector of some type on which to send it. A trivial example is the `http_get.py` script which uses a `TCP4CONNECTOR` to contact a web server, execute a simple *GET* command, and print the results.

Although everything that is done in the `http_get` script could be done far better with Python's native HTTP classes the script does show how easy it is to set up a connector.

For the purposes of protocol development and testing it is more interesting to look at the `PCAPCONNECTOR` class, which is used to read and write raw packets to the network. Figure 15 shows a section of the `icmpv4test` test script which transmits an ICMPv4 echo, aka ping, packet.

¹

The `test_icmpv4_ping()` function contains a good deal of code but we are only concerned with the last two lines at the moment. The next to the last line opens a raw pcap socket on the localhost, lo0, interface which allows us to write packets directly to that interface. The last line writes a packet to the interface. We will come back to this example again in section 9.

¹Note that on most operating system you need root privileges in use the `PCAPCONNECTOR` class.

```

def test_icmpv4_ping(self):
    ip = ipv4()
    ip.version = 4
    ip.hlen = 5
    ip.tos = 0
    ip.length = 84
    ip.id = 1
    ip.flags = 0
    ip.offset = 0
    ip.ttl = 33
    ip.protocol = IPPROTO_ICMP
    ip.src = 2130706433
    ip.dst = 2130706433

    icmp = icmpv4()
    icmp.type = 8
    icmp.code = 0
    icmp.cksum = 0

    echo = icmpv4echo()
    echo.id = 32767
    echo.seq = 1

    lo = localhost()
    lo.type = 2
    packet = Chain([lo, ip, icmp, echo])

    input = PcapConnector("lo0")
    input.setfilter("icmp")

    output = PcapConnector("lo0")
    out = output.write(packet.bytes, 88)

```

Figure 15: Transmitting a Raw Ping Packet

```

import pcs
from pcs.packets.ethernet import ethernet

def main():

    from optparse import OptionParser

    parser = OptionParser()
    parser.add_option("-i", "--interface",
                      dest="interface", default=None,
                      help="Which interface to snarf from.")

    (options, args) = parser.parse_args()

    snarf = pcs.PcapConnector(options.interface)

    while 1:
        packet = ethernet(snarf.read())
        print packet
        print packet.data

main()

```

Figure 16: Packet Snarfing Program

8 Receiving Packets

In order to receive packets we again use the `CONNECTOR` classes. Figure 16 shows the simplest possible packet sniffer program that you may ever see.

The **snarf.py** reads from a selected network interface, which in this case must be an Ethernet interface, and prints out all the Ethernet packets and *any upper level packets that PCS knows about*. It is this second point that should be emphasized. Any packet implemented in **PCS** which has an upper layer protocol can, and should, implement a `next()` method which correctly fills in the packet's `data` field with the upper level protocol. In this case the upper layer protocols are likely to be either ARP, IPv4 or IPv6, but there are others that are possible.

9 Chains

We first saw a the `CHAIN` class in Figure 15 and we'll continue to refer to that figure here. CHAINS are used to connect several packets together, which allows use to put any packet on top of any other. Want to transmit an Ethernet packet on top of ICMPv4? No problem, just put the Ethernet packet after the ICMPv4 packet in the chain. Apart from creating arbitrary layering, CHAINS allow you to put together better known set of packets. In order to create a valid ICMPv4 echo packet we need to have a IPv4 packet as well as the proper framing for the localhost interface. When using **pcap** directly even the localhost interface has some necessary framing to indicate what type of packet is being transmitted over it.

The packet we're to transmit is set up as a `CHAIN` that contains four other packets: localhost, IPv4, ICMPv4, and Echo. Once the chain is created it need not be static, as in this example, as changes to any of the packets it contains will be reflected in the chain. In order to update the actual bytes the caller has to remember to invoke the `encode()` method after any changes to the packets the chain contains.²

10 Displaying Packets

To be done, to be done...

²This may be fixed in a future version to make CHAINS more automatic.