Packet Construction Set

George V. Neville-Neil

February 18, 2014

Contents

Abstract

We had Ethernet headers, IP packets, TCP segments, a gaggle of HTTP requests and responses, also UDP, NTP, and DHCP. Not that we needed all that just to communicate but once you get locked into a serious packet collection the tendency is to push it as far as you can. - Deepest apologies to Hunter S. Thompson

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.

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 (octect) in length (Version, IHL, Flags)

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

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

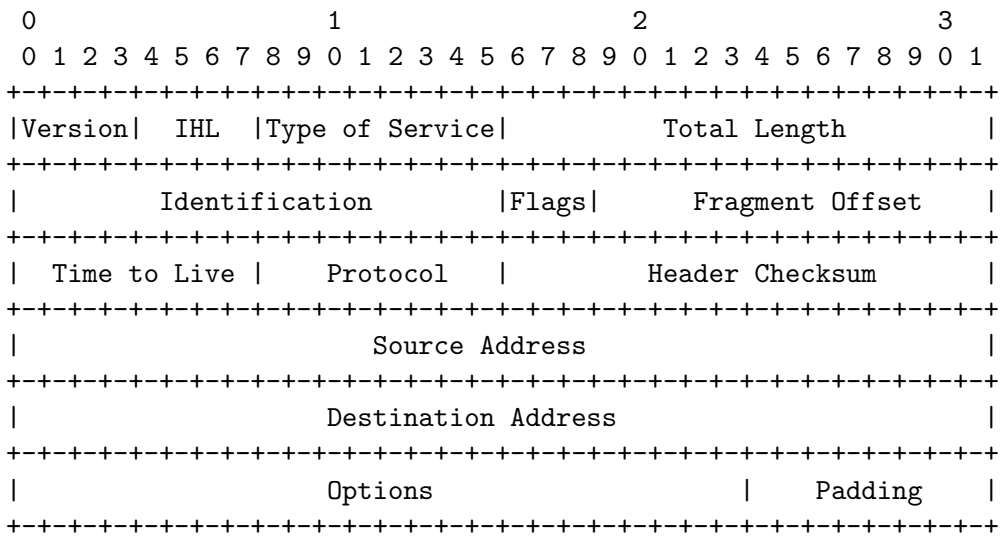

Figure 1: IPv4 Header Format

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 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.

```
>>> 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\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\x00\x00'
```

Figure 3: The `bytes` Field of the Packet

```
1 ip = ipv6()
2 assert (ip != None)
3 ip.traffic_class = 1
4 ip.flow = 0
5 ip.length = 64
6 ip.next_header = 6
7 ip.hop = 64
8 ip.src = inet_pton(AF_INET6, "::1")
9 ip.dst = inet_pton(AF_INET6, "::1")
```

Figure 4: IPv6 Class

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 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 Figure7.

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

Figure 5: Bytes of the IPv6 Packet

```
1 >>> print ip
2 version 6
3 traffic_class 1
4 flow 0
5 length 64
6 next_header 6
7 hop 64
8 src ::1
9 dst ::1
```

Figure 6: Printing a Packet

```
>>> 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 8. Thoughout this section we will use the example of a network layer packet, IPv6, and a packet about the transport layer, DNS. Using both low and high level packets should give the reader a good feel for how to add most of the packets they would be expected to work with.

The code in Figure **??** defines a new class, one that will describe an IPv6 packet, sub-classed from the PACKET base class. There are a small number of reserved field names that you *must not* use when defining your packets. For reference all of the reserved field names are given in Table 9 and most of them will also be discussed in this section. Resereved names that are part of a class, as opposed to an object, are preceeded by an underscore, _, to further set them apart. Do not use underscores to start your fieldnames. *You have been warned!*

Each packet class in PCS is defined in a similar way. After sub-classing from the PACKET class, there should be a Python style text string describing the class. The fields are defined next,as shown on lines 10 through 17, in the order in which they are stored in the packet. Various types of fields are supported by PCS and they are all covered in Section **??**. After all of the fields have been listed, the PACKET class's __init__ method is called with

```
1  class ipv6(pcs.Packet):
2      """A class that contains the IPv6 header.  All other data is
3  chained on the end."""
4
5      _layout = pcs.Layout()
6      _map = None
7
8      def __init__(self, bytes = None):
9          """IPv6 Packet from RFC 2460"""
10         version = pcs.Field("version", 4, default = 6)
11         traffic = pcs.Field("traffic_class", 8)
12         flow = pcs.Field("flow", 20)
13         length = pcs.Field("length", 16)
14         next_header = pcs.Field("next_header", 8, discriminator=True)
15         hop = pcs.Field("hop", 8)
16         src = pcs.StringField("src", 16 * 8)
17         dst = pcs.StringField("dst", 16 * 8)
18         pcs.Packet.__init__(self,
19                             [version, traffic, flow, length, next_header,
20                              src, dst], bytes)
21         self.description = "IPv6"
22
23         self._map = ipv6_map.map
24
25         if (bytes != None):
26             ## 40 bytes is the standard size of an IPv6 header
27             offset = 40
28             self.data = self.next(bytes[offset:len(bytes)])
29         else:
30             self.data = None
```

Figure 8: IPv6 Packet Class

Field Name	Use
`_layout`	Used to store the layout of the packet
`_map`	Used to demultiplex higher layer packets
`next`	A method used to unencapsulate higher layer packets
`bytes`	Storage for the raw bytes of the packet
`data`	Pointer to the next higher layer packet
`description`	Textual description of the packet

Figure 9: Reserved Fields and Methods in PCS

Name	Use	Initialiazation Arguments
Field	Abitrary Bit Field	Name, Width in Bits, Default Value
StringField	String of Bytes	Name, Width In Bits, Default Value
LengthValueField	A set of Values with Associated Lengths	Name, Width in Bytes, Default Value

Figure 10: Fields Supported by PCS

three arguments. The SELF object, an array of the fields, in the order in which they will appear in a packet, and the `bytes` variable that was passed to the packet object's `init` method. Once the packet is initalized we set its description, on line 20.

Any packet that may contain data at a higher layer, such as a network packet will then use its `next` method to unencapsulate any higher layer packets. On lines 25 through 30 the `init` method attempts to unencapsulate any data after the header itself. Every packet object either has a valid `data` or it is set to `None`. Higher level programs using PCS will check for `data` being set to `None` in order to know when they have reached the end of a packet so it must be set correctly by each packet class. The `next` method used here is from the PACKET base class but it can also be overridden by a programmer, and this is done in the TCP class which can be found in `pcs/packets/tcp.py`.

4.1 Working with Different Types of Fields

Part of packet initialization is to set up the fields that the packet will contain. Fields in PCS are objects in themselves and they are initialized in different ways, depending on their type. A brief list of the currently supported fields is given in Table 10.

Each field has several possible arguments, but the two that are required are a name, which is the string field specified as the first argument and

a width, which is the second argument. Note that some field widths are specified in *bits* and some in *bytes* or

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

It would have been convenient if all network protocol packets were simply lists of fixed length fields, but that is not the case. PCS defines two extra field classes, the StringField and the LengthValueField.

The StringField is simply a name and a width in bits of the string. The data is interpreted as a list of bytes, but without an encoded field size. Like a Field the StringField has a constant size.

Numerous upper layer protocols, i.e. those above UDP and TCP, use length-value fields to encode their data, usually strings. In a length-value field the number of bytes being communicated is given as the first byte, word, or longword and then the data comes directly after the size. For example, DNS [**?**] encodes the domain names to be looked up as a series of length-value fields such that the domain name pcs.sourceforge.net gets encoded as 3pcs11sourceforge3net when it is transmitted in the packet.

The LengthValueField class is used to encode length-value fields. A LenghtValueField has three attributes, its name, the width in bits of the length part, and a possible default value. Currently only 8, 16, and 32 bit fields are supported for the length. The length part need never been set by the programmer, it is automatically set when a string is assigned to the field as shown in 11.

Figure 11 shows both the definition and use of a LengthValueField. The definition follows the same system as all the other fields, with the name and the size given in the initialization. The dnslabel class has only one field, that is the name, and it's length is given by an 8 bit field, meaning the string sent can have a maximum length of 255 bytes.

When using the class, as mentioned, the size is not explicitly set. One last thing to note is that in order to have a 0 byte terminator the programmer assigns the empty string to a label. Using the empty string means that the length-value field in the packet has a 0 for the length which acts as a terminator for the list. For a complete example please review `dns_query.py` in the `scripts` directory.

4.2 Built in Bounds Checking

One of the nicer features of PCS is built in bounds checking. Once the programmer has specified the size of the field, the system checks on any attempt to set that field to make sure that the value is within the proper

```
1  class dnslabel(pcs.Packet):
2      """A DNS Label."""
3
4      layout = pcs.Layout()
5
6      def __init__(self, bytes = None):
7          name = pcs.LengthValueField("name", 8)
8          pcs.Packet.__init__(self,
9                              [name],
10                             bytes = bytes)
11
12         self.description = "DNS Label"
13
14 ...
15
16     lab1 = dnslabel()
17     lab1.name = "pcs"
18
19     lab2 = dnslabel()
20     lab2.name = "sourceforge"
21
22     lab3 = dnslabel()
23     lab3.name = "net"
24
25     lab4 = dnslabel()
26     lab4.name = ""
```

Figure 11: Using a LengthValueField

```
1  >>> from pcs.packets.ipv4 import *
2  >>> ip = ipv4()
3  >>> ip.hlen = 16
4  Traceback (most recent call last):
5  [...]
6  pcs.FieldBoundsError: 'Value must be between 0 and 15'
7  >>> ip.hlen = -1
8  Traceback (most recent call last):
9  [...]
10 pcs.FieldBoundsError: 'Value must be between 0 and 15'
11 >>>
```

Figure 12: Bounds Checking

bounds. For example, in Figure 12 an attempt to set the value of the IP packet's header length field to 16 fails because the header length field is only 4 bits wide and so must contain a value between zero and fifteen.

PCS does all the work for the programmer once they have set the layout of their packet.

4.3 Decapsulating Packets

One of the key concepts in networking is that of encapsulation, for example an IP packet can be encapsulated in an Ethernet frame. In order to provide a simple way for programmers to specifying the mapping between different layers of protocols PCS provides a next method as part of the PACKET base class. There are a few pre-requisites that the programmer must fulfill in order for the next method to do its job. The first is that at least one FIELD must be marked as a *discriminator*. The descriminator field is the one that the next method will use to decapsulate the next higher layer packet. The other pre-requisite is that the programmer define a mapping of the discriminator values to other packets. An example seems the best way to make sense of all this.

Figure 13 shows an abbreviated and combined listing of the ETHERNET class and its associated mapping class. The full implementation can be found in the source tree in the files `pcs/packets/ethernet.py` and `pcs/packets/ethernet_map.py` respectively. On line 7 a class variable, one that will be shared across all instances of this object, is created and set to the map that is defined in the ethernet_map module.

```
 1 import pcs
 2 import ethernet_map
 3
 4 class ethernet(pcs.Packet):
 5
 6     _layout = pcs.Layout()
 7     _map = None
 8
 9     def __init__(self, bytes = None):
10         """initialize an ethernet packet"""
11         src = pcs.StringField("src", 48)
12         dst = pcs.StringField("dst", 48)
13         type = pcs.Field("type", 16, discriminator=True)
14         etherlen = 14
15
16         pcs.Packet.__init__(self, [dst, src, type], bytes = bytes)
17         self.description = "Ethernet"
18
19         self._map = ethernet_map.map
20
21         if (bytes != None):
22             self.data = self.next(bytes[etherlen:len(bytes)])
23         else:
24             self.data = None
25 ...
26
27 import ipv4, ipv6, arp
28
29 ETHERTYPE_IP            = 0x0800        # IP protocol
30 ETHERTYPE_ARP           = 0x0806        # Addr. resolution protocol
31 ETHERTYPE_IPV6          = 0x86dd        # IPv6
32
33 map = {ETHERTYPE_IP: ipv4.ipv4,
34        ETHERTYPE_ARP: arp.arp,
35        ETHERTYPE_IPV6: ipv6.ipv6}
```

Figure 13: The Ethernet Packet and Mapping Classes

The actual mapping of discriminators to higher layer packets is done in the mapping module. Line 27 shows the mapping module importing the higher layer objects, in this case the IPV4, IPV6, and ARP packets which can be encapsulated in an Ethernet frame. The map is really a `Python` dictionary where the key is the value that the `next` method expects to find in the field marked as a *discriminator* in the ETHERNET packet class. The values in the dictionary are packet class constructors which will be called from PCS's PACKET base class.

If the preceeding discussion seems complicated it can be summed up in the following way. A packet class creator marks a single FIELD as a *discriminator* and then creates a mapping module which contains a dictionary that maps a value that can appear as a discriminator to a consctructor for a higher layer packet class. In the case of Ethernet the discriminator is the `type` field which contains the protocol type. An Ethernet frame which contains an IPv4 packet will have a `type` field containing the value 2048 in decimal, $0x800$ hexadecimal. The PACKET base class in this case will handle decapsulation of the higher layer packet.

Mapping classes exist now for most packets, although some packets, such as TCP and UDP, require special handling. Refer to the `next` method implementations in `pcs/packets/tcp.py` and `pcs/packets/udp.py` for more information.

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 though 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 Figure14 we see two different save files being opened. The first,

```
>>> 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 14: Determining the Bottom Layer

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

Figure 15: Finding the Datalink Offset

`etherping.out` is a tcpdump file that contains data collected on an Ethernet interface, type `DLT_EN10` and the second, `loopping.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 of the PCAP class gives the data link offset. Figure15 continues the example shown in Figure14 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

```
>>> 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 16: Reading in a Packet

garbage.

With all this background we can now read a packet and examine it. Figure 16 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.

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 understand 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 Figure17 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.

```
>>> 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,
>>> 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 17: Packet Decapsulation on Read

```
1  import pcs
2
3  from socket import *
4
5  def main():
6
7      conn = pcs.TCP4Connector("127.0.0.1", 80)
8      conn.write("GET / \n")
9      result = conn.read(1024)
10
11     print result
12
13 main()
```

Figure 18: HTTP Get Script

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 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 19 shows a section of the `icmpv4test` test script which transmits an ICMPv4 echo, aka ping, packet.*

[1]

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

```
1       def test_icmpv4_ping(self):
2           ip = ipv4()
3           ip.version = 4
4           ip.hlen = 5
5           ip.tos = 0
6           ip.length = 84
7           ip.id = 1
8           ip.flags = 0
9           ip.offset = 0
10          ip.ttl = 33
11          ip.protocol = IPPROTO_ICMP
12          ip.src = 2130706433
13          ip.dst = 2130706433
14
15          icmp = icmpv4()
16          icmp.type = 8
17          icmp.code = 0
18          icmp.cksum = 0
19
20          echo = icmpv4echo()
21          echo.id = 32767
22          echo.seq = 1
23
24          lo = localhost()
25          lo.type = 2
26          packet = Chain([lo, ip, icmp, echo])
27
28          icmp_packet = Chain([icmp, echo])
29          icmp.checksum = icmp_packet.calc_checksum()
30
31          packet.encode()
32
33          input = PcapConnector("lo0")
34          input.setfilter("icmp")
35
36          output = PcapConnector("lo0")
37          out = output.write(packet.bytes, 88)
```

Figure 19: Transmitting a Raw Ping Packet

```
1  import pcs
2  from pcs.packets.ethernet import ethernet
3
4  def main():
5
6      from optparse import OptionParser
7
8      parser = OptionParser()
9      parser.add_option("-i", "--interface",
10                       dest="interface", default=None,
11                       help="Which interface to snarf from.")
12
13     (options, args) = parser.parse_args()
14
15     snarf = pcs.PcapConnector(options.interface)
16
17     while 1:
18         packet = ethernet(snarf.read())
19         print packet
20         print packet.data
21
22 main()
```

Figure 20: Packet Snarfing Program

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.

8 Receiving Packets

In order to receive packets we again use the CONNECTOR classes. Figure 20 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 19 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]

CHAINS can also calculate RFC 792 style checksums, such as those used for ICMPv4 messages. The checksum feature was used in Figure 19. Because it is common to have to calculate checksums over packets it made sense to put this functionality into the CHAIN class.

10 Displaying Packets

```
To be done, to be done...
```

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