COMPSCI 314 Lab: 2010-S1

Department of Computer Science
The University of Auckland

DongJin Lee, Nevil Brownlee, Brian Carpenter, Habib Naderi
{dongjin, n.brownlee, brian}@cs.auckland.ac.nz, hnad002@aucklanduni.ac.nz

Version 1.1 [May-2010]

I. Introduction

In this guideline, you are to learn to capture packets and analyze them using some tools. This guideline should be trivial and thus we will only cover basic materials you'd need to know to start the assignment. You will learn to use *windump* and *Wireshark*. While the guideline here should be sufficient, you *should* also read other web resources so as to better understand them. We strongly advise you to revise CS215, and we *expect* you to study on your own any material that you might not be familiar with. In particular, you need to be familiar with the TCP/IP concepts you've learned in CS215.

II. Course book chapters

'Understanding Data Communications and Networks (3e)' by William A. Shay

The chapters below are highly recommended, although not all are covered in this guideline.

- Chapter 1.3 to 1.4 (page 15 to 43)
- Chapter 9.1 to 9.2 (page 396 to 399)
- Chapter 9.3 (page 410 to 416)
- Chapter 10.1 to 10.3 (page 462 to 469)
- Chapter 11.1 to 11.2 (page 524 to 528, 537 to 539, 541 to 549)
- Chapter 11.4 (page 571 to 576)

III. Brief review of networking concepts

The lowest level in any communication system is the most likely either wired or wireless. For 'wire' communications, signals are carried by electricity (e.g., on copper wires) or light (e.g., on glass fibres). For wireless communications, they are carried by radio waves. We call this lowest level layer-1, the physical layer in the OSI or TCP/IP models. The way signals carry binary information is vastly different depending on the types of medium with different standards. As long as both end-points use the same standard, we shouldn't need to worry. If a device is said to work only at layer-1, then you should know that it will only work on the 'bit' level, i.e., physical level. For example, typical 'hubs' or 'repeaters' would be regarded as layer-1 devices, as they simply regenerate (or amplify) an incoming signal to all outgoing ports.

It is important to know that when the bits are sent, there are certain 'gaps' to indicate the end of data. For 10Mb/s Ethernet, there is at least 9μs of an idle period for each frame. Thus, you would observe this amount of gap between the frames. These gaps allow a receiver to prepare (or synchronize) for the next frame. Failing to provide the gaps can cause the receiver to discard frames as 'unreadable'.

Each frame or datagram should contain sufficient information to be delivered across the network. In that sense, they contain 'header' and 'payload'. The headers contain vital information such as destination address, types of payload, and so on. Once data is safely delivered, headers are no longer required (so they are called 'overheads'). Payloads contain the actual 'data', but data can also include headers (as well as data). This may be confusing, however taking an example of your favorite MP3 file, you may call it data, but from the structure point of view, the file content itself would contain some header information such as bit-rates, ID tags and so on. You would observe such examples almost everywhere! It is only important to distinguish header/payload depending on the *context*. If you are working on the file itself, such as reducing the ID tag sizes, then you would divide the file into two types: a header for ID tags, and data for MP3 signals. However, if you are sending this file over the network, then the file itself is regarded as a payload: it would be split into pieces, each piece encapsulated by layers of (network-specific) headers.

IV. Brief TCP/IP Model concepts

A **frame** operates at layer-2 (datalink layer), a **packet** operates at layer-3 (network layer) and **TCP/UDP** operates at layer-4 (transport layer). Under normal network setups, this means that your user-data (often regarded as an application layer) is encapsulated by the transport layer, then again encapsulated by the network layer, and finally by the datalink layer.

To simply the explanation, Figure 1 shows how your user-data is carried across the network. It is important to know that each layer operates *independently*, thus communicating devices often disregard upper layers. For example, switches will only look at the header of frames (i.e., frame/MAC addresses) and regard the rest as 'payload'. Routers observe *more*, by taking out the frame payload to look for the packet header (e.g., IP address) and regard the rest as 'payload'. Your machine will interpret the rest. For example, all TCP states are controlled by your Operating System's IP stack.

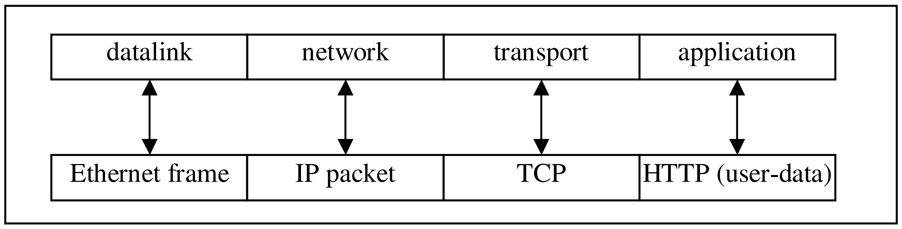


Figure 1. Top blocks show the layers of TCP/IP model and bottom blocks show a popular usage example.

For example, assuming you are connected to a network with Internet access, and visiting some external websites, a user-data (e.g., HTTP request) created by your machine will be encapsulated by the TCP, IP, and frame before it travels through the network (e.g., switches). The switches will forward your frame to, for instance, the next closest switch/router. Note that, while a **frame** is travelling between the switches, its source and destination address could change (but its upper layers are preserved, such as your IP/TCP packet). Once a frame enters the router, it will *strip* off the frame header, look at the IP packet (observing destination IP address), and forward the packet to the next (best) router, and so on. The important distinction here is that your source and destination IP addresses are *unchanged* throughout the communication. The rest of the upper layers (e.g., TCP) are handled by your machine, not the switches or routers! Obviously, there are some exceptions, such as NAT enabled routers that could change the source/destination IP and TCP addresses.

There are many vague words used in networking literature. That is, some terminologies are *inconsistent* between the books and papers! One may use 'frame' to indicate 'packet' and others do vice versa. Often when exchanging data across the network, people say *'...sending and receiving packets'*, but actually they mean *'…sending and receiving frames'*. To be even more precise, ultimately, one could say *'…sending and receiving binary/bits/signals'*. Generally, you should attempt to understand overall concepts as these words are normally used in a specific context (e.g., one may choose to use 'bits/signals' to explain how signals travel on the wire).

Further, there may be several names for similar terms. For example, 'address' can be explained as 'frame address', 'packet address', 'MAC address', 'IP address', 'TCP address', or UDP address'! To be less confusing, one chooses to be more specific, e.g., 'frame address'. However, this is also similar to a 'MAC address' since its 48bit MAC addressing scheme (OUI+NIC) is used as the frame address. Also, 'packet address' or 'IP address' mean the same. As for TCP/UDP address, they are usually called a 'port number'.

There are just too many protocols described in networking materials. Fortunately, only a few protocols are being used widely. Before you begin to learn and use the tools, we expect you to have a *brief* grasp of following protocols: IP, TCP, UDP, ICMP, ARP and DHCP.

V. Brief overview of the tools

A. *tcpdump / windump / packet capture library*

tcpdump is one of the most popular tools for observing and analyzing network packets, and this tool runs on Unix/Linux machines. Here, we are using a Windows version called *windump*. Both versions have an underlying API library called *pcap* (*libpcap* in Linux, and *winpcap* in Windows). Often people say *tcpdump* to refer both versions. Both *tcpdump* and *windump* are non-GUI, meaning that you use these tools from a command line. We will discuss this further in Section VI. While some little differences may exist, you do not need to worry about the details. What you need to be aware of is that once you've acquired either of them, you can start capturing packets!

Some of our CS Lab machines now run Windows 7 while some other still run Windows Vista, and we have installed *winpcap* on all of them. This means that once you are enrolled in CS314, you should be able to run this tool. We do not have the Linux version and thus you are limited to capture packets under Windows only! You can of course, install Linux versions on your own machine. Below are the links that you can download them from. Also, explore the websites as you will find many useful features and hints, for example, FAQ sections.

http://www.tcpdump.org/ for linux users
http://www.winpcap.org/ for windows users
http://www.winpcap.org/misc/faq.htm Frequently Asked Questions

For a simple outline, you can also refer to *wikipedia*:
http://en.wikipedia.org/wiki/Tcpdump
http://en.wikipedia.org/wiki/Pcap
http://en.wikipedia.org/wiki/Packet_capturing

B. *Wireshark*

This is an advanced toolkit that incorporates a GUI with various useful features. The original version (*Ethereal*) has been replaced by *Wireshark*, so we will use Wireshark rather than *Ethereal*. Both versions are the same except that the name has changed, and Ethereal is no longer updated. Unlike windump which merely provides useful information about the packets, Wireshark can interpret almost every packet, for instance, types/contents/sizes. Also, it can give you a *detailed* summary of the captured packets, all within a GUI.

http://www.wireshark.org/ for both Windows and Linux users

There are little tradeoffs between the two tools. Generally, Wireshark is more preferable than windump as far as the usage and functionalities are considered. Figure 2 shows a simple diagram of how pcap and the tools are related.

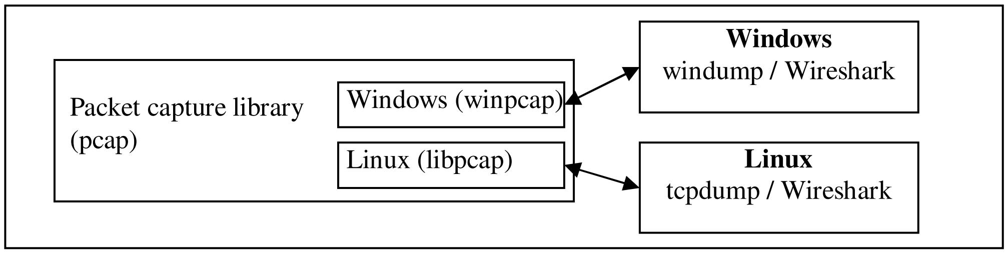


Figure 2. Both *windump* and *Wireshark* uses pcap library

Note that you are only capturing packets on your lab machine! It's Network Interface Card (NIC) will not be able to 'hear' packets from other machines except some broadcast messages. This is because the machines are connected to a switch (Also, we do not recommend you eavesdropping messages of other users as this could breach University policies). However, if you are an administrator and have multiple networked machines at your own residence, you can set up the network in such a way to allow your machine to hear *all* packets going in/out from Internet (and your local network). This is relatively straightforward if you have a hub; simply connect all machines to it including yours, and run the tool. As mentioned, the hub is a layer-1 device that simply regenerates the signals. This approach has some disadvantages because the machines can suffer from the hub bottleneck and hubs are rather rare nowadays with switches being more common.

There are other solutions, for example, a 'managed switch' can be configured to set one or more port to perform 'port mirroring' (also called span-port). This sets the switch to copy and forward any incoming/outgoing packet to the mirrored port. Thus, a machine connected to that port will be able to hear all packets in the network.

As for our typical university network, there are over 15,000 networked hosts in our campus! All incoming/outgoing packets are passing through a single 1Gb/s channel with backup links, and our system can observe the packets at this point. There are countless benefits in capturing and measuring the packets, such as billing/accounting (NetAccount), detecting malicious (DDoS) traffic, bandwidth provisioning, and so on. In other words, *'…if you don't measure your network, you don't know what's happening…'*

If you are interested in setting up your own network with monitoring capabilities, you may find this link useful. http://wiki.wireshark.org/CaptureSetup/Ethernet

Before you start using these tools, first create a blank directory, and make sure you have some storage space in your AFS. You can check your space by **right clicking** on your AFS Drive (e.g., **H:**), select **Volume/Partition** to select **Properties**. As a rule of thumb, have at least 50 to 100MB of free space in your AFS.

VI. *windump usage*

We will first start with **windump**. Again, this is a Command Line Input (CLI) tool, which does not have any GUI front-end. All option parameters are Unix-style where you type hyphen (**–**) with a letter. Here, you need to learn some of the command to capture the packets. As you can see from the output below, there are many arguments/options! Fortunately you do not need to learn all the commands as we will only cover the basics.

```
D:\>windump -help
windump version 3.9.5, based on tcpdump version 3.9.5
WinPcap version 4.0 (packet.dll version 4.0.0.755), based on libpcap version 0.9.5
Usage: windump [-aAdDeflLnNOpqRStuUvxX] [ -B size ] [-c count] [ -C file_size ]
                [ -E algo:secret ] [ -F file ] [ -i interface ] [ -M secret ]
                [ -r file ] [ -s snaplen ] [ -T type ] [ -w file ]
                [ -W filecount ] [ -y datalinktype ] [ -Z user ]
                [ expression ]
```

Here are the lists of the options that you need to learn. **–B, –c, –i, –e, –r, –n, –s, –w**
We will briefly explain with an example. For those wanting more detailed explanations, you can go to the link here. http://www.winpcap.org/windump/docs/manual.htm

We specify which NIC we want to capture packets from. You can view a list of interfaces by adding **–D** to check which one you want to monitor.

```
D:\>windump -D
1.\Device\NPF_{6A81D585-1844-4B46-B2B1-E471A96173DF} (MS LoopBack Driver)
2.\Device\NPF_{9F23AED8-893D-486B-9B3F-53BFCA7DAA06} (Realtek)
3.\Device\NPF_{12F01666-2E62-40F5-88AD-999A67176997} (VMware Virtual Ethernet Adapter)
4.\Device\NPF_{12FD419D-C0A8-40E3-B741-19468FFFBE11} (MS Tunnel Interface Driver)
5.\Device\NPF_{8082E9B6-DEF0-40E8-B012-923E2EEB67EF} (VMware Virtual Ethernet Adapter)
```

What you see above is likely to be different in your machine. So you should just concern about the actual 'ethernet' NIC. In the above example, it is number **2**. We specify the interface number by adding **–i2**.

```
D:\>windump -i2
windump: listening on \Device\NPF_{9F23AED8-893D-486B-9B3F-53BFCA7DAA06}
0 packets captured
0 packets received by filter
0 packets dropped by kernel
```

The above example did not capture any packet, because it was immediately stopped (CTRL+C). The first (**captured**) and second (**received by filter**) lines should have equal counts. There are some differences between the first and second line. For example, if you are capturing the packets outgoing to some specific address, then this first line will show the number of packets actually captured. The second line shows the (total) number of packets sent/received by the NIC. The third line shows how many packets were 'dropped'. Packet drops can occur for a variety of reasons, such as lack of processor, capacity or disk writing speed. Since you are only capturing packets on your local machine, there *should* be zero packet drops. By default, windump uses 1MB of buffer which can be changed by adding **–B** followed by the amount of buffer in kB. Also, windump will attempt to resolve any IP addresses to their names (by requesting DNS lookup). This process often slows down the packet capturing and may cause some packet drops. You should use **–n** to disable any name resolution.

```
D:\>windump –i2 –n
```

One of the simplest ways to confirm whether you are capturing correctly, is to send/receive packets on your NIC. This can be easily done by 'surfing a website'. For example, going to the main page of the university home page would result in capturing over 100 packets. You will have to stop the windump to see what has been captured. Note that you can specify to stop after capturing some number of packets by adding **–c** followed by packet counts. Whenever there are many packets being captured, it is often very hard to observe packets that you are interested in. For instance, running the above command would be likely to capture broadcast packets as well.

Here, as a simple example, we want to capture ICMP packet exchanges between the university home page and your machine. We can find the university homepage IP address using the **nslookup** command, shown below.

```
D:\>nslookup www.auckland.ac.nz
Server:  kronos1.cs.auckland.ac.nz
Address:  130.216.35.35:35

Name:    www-vip.auckland.ac.nz
Address:  130.216.11.141
Aliases:  www.auckland.ac.nz
```

Often you do not need to know the IP address you are monitoring, since most of the tools will automatically convert name-to-IP by default. Here, we tell windump to capture only packets going to/from university homepage. Then, we open a new command prompt and use the **ping** command to send ICMP packets to the university homepage. So in your first command prompt,

type: `windump –i2 –n host www.auckland.ac.nz` and in your second command prompt,

type: `ping www.auckland.ac.nz`

First command prompt

```
D:\>windump –i2 –n host www.auckland.ac.nz
windump: listening on \Device\NPF_{9F23AED8-893D-486B-9B3F-53BFCA7DAA06}
17:05:59.169380 IP 130.216.37.108 > 130.216.11.141: ICMP echo request, id 1, seq 12715, length 40
17:05:59.169668 IP 130.216.11.141 > 130.216.37.108: ICMP echo reply, id 1, seq 12715, length 40
17:06:00.162740 IP 130.216.37.108 > 130.216.11.141: ICMP echo request, id 1, seq  12716, length 40
17:06:00.162957 IP 130.216.11.141 > 130.216. 37.108: ICMP echo reply, id 1, seq 12716, length 40
17:06:01.162908 IP 130.216.37.108 > 130.216. 11.141: ICMP echo request, id 1, seq  12718, length 40
17:06:01.163119 IP 130.216.11.141 > 130.216.37.108: ICMP echo reply, id 1, seq 12718, length 40
17:06:02.163007 IP 130.216.37.108 > 130.216.11.141: ICMP echo request, id 1, seq 12719, length 40
17:06:02.163205 IP 130.216.11.41 > 130.216.37.108: ICMP echo reply, id 1, seq 12719, length 40

8 packets captured
140 packets received by filter
0 packets dropped by kernel
```

Second command prompt

```
D:\>ping www.auckland.ac.nz

Pinging www-vip.auckland.ac.nz [130.216.11.141] with 32 Bytes of data:

Reply from 130.216.11.141: Bytes=32 time<1ms TTL=61
Reply from 130.216.11.141: Bytes=32 time<1ms TTL=61
Reply from 130.216.11.141: Bytes=32 time<1ms TTL=61
Reply from 130.216.11.141: Bytes=32 time<1ms TTL=61

Ping statistics for 130.216.11.141:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

As you can see from the above example, we capture the packets that are either going to/from www.auckland.ac.nz and we intentionally ping (send ICMP Request packets) to that host. In total, eight packets have been captured (and total 140 packets are observed). To better understand the output lines of windump, we will explain the first two. At time `17:05:59.169380`, an IP packet (`ICMP echo request`) from IP address `130.216.37.108` was sent to IP address `130.216.11.141`, and vice versa for the second line (`ICMP echo reply`). Notice that the reply ICMP packet's ID and sequence number is the same as the first line, indicating that the response is to the initial request. Further, `length 40` indicates that the ICMP packet size is 40 bytes, but the full *frame* (captured) size is 74 bytes. This is because of the frame header (14) + IP header (20) + ICMP header (8) + Payload (32) = 74. As well, you can find the ping RTT more precisely by differencing the two times, i.e., `.169668 - .169380 = 0.000288 = 0.288ms`. (Note you can do this by adding **–ttt** to display the time differences for consecutive packets). An ICMP packet's payload does not contain much meaningful data. Nevertheless, to print out what is being captured in ASCII format, you can add **–A** which will display the contents of each packet in ASCII. This can be useful if the packets contain typical HTML objects.

Often network administrators store the captured packets into files for later analysis. Here, we have not yet saved these captured packets into disk storage. To do so, you need to add **–w** followed by a path/directory and file name.

```
D:\>windump -i2 -waaa.bpf -n host 130.216.11.141
```

This command will save the captured packets into a file called `aaa.bpf` instead of printing them out. Although a file extension name could be different, windump uses a common format, known as Berkeley Packet Filter. This file is often called an *offline* traffic data trace or simply a *trace file*. You can also read back such saved files by replacing **–i2** with **–r** followed by the file's path/directory and file name.

```
D:\>windump -raaa.bpf -n
```

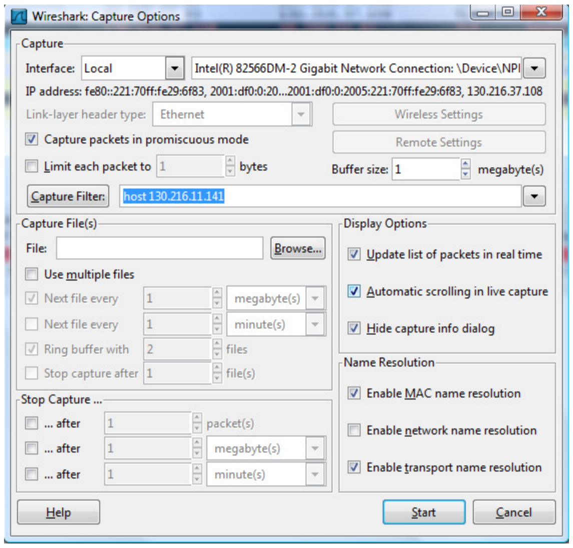
You need to be aware of the fact that capturing and saving packets can require a large amount of storage. For example, capturing packets from a gateway link transferring at 20-30MB/s may sound a little, but producing a trace file for a whole day can easily fill up the disk space, e.g., it would require at least 1.7TB. Also, disk storage may be a bottleneck if it cannot write faster than the data rate observed on the network link, resulting packet drops. Furthermore, CPU and disk usage is higher when capturing a *full* payload of the packets, which can also cause packet drops. For these reasons, full payload traces are rarely used. Instead, it is possible to just capture the packet *headers* and still obtain vital network information. To capture a limited number of bytes for each packet, you need to add **–s** followed by the number of bytes.

```
D:\>windump -i2 -s4 -waaa.bpf -n host 130.216.11.141
```

This example captures only the first *four* bytes of each frame. This may well reduce the amount of disk storage required, but the loss of packet details is significant. Reading back the trace file, you will not be able to learn much from what you've captured (e.g., are they ICMP packets?). Here, you would need to choose a more appropriate (larger) snap length.

VII. *Wireshark usage*

In this section, we will use *Wireshark* to capture packets (as we did with windump), then observe and analyze some of the captured packets, all in GUI! Configuring the Wireshark is relatively simple with GUI setups. Once the program has started, go to **Capture** and **Options** to bring up **Capture Options**. Here, select your Ethernet NIC and set **Capture Filter** (host 130.216.11.41) as shown in Figure 3. There are several other options you can enable/disable, such as disabling MAC name resolution, and limiting the packet capturing size. But for now, select **Start**. To test whether it is working or not, you can send some packets, e.g., using a ping command in the command prompt. By default, Wireshark will immediately display captured-packets similar to windump. Figure 4 shows eight ICMP packets captured after the ping command. We also advise you to stop the capturing process whenever you need to observe or analyze packets. Similar to windump, you can save the captured packets into a trace file; the command to do this is under the **File** menu.



Wireshark: Capture Options

Capture
Interface: Local — Intel(R) 82566DM-2 Gigabit Network Connection: \Device\NPI
IP address: fe80::221:70ff:fe29:6f83, 2001:df0:0:20...2001:df0:0:2005:221:70ff:fe29:6f83, 130.216.37.108
Link-layer header type: Ethernet
[✓] Capture packets in promiscuous mode
[ ] Limit each packet to 1 bytes
Buffer size: 1 megabyte(s)
Capture Filter: host 130.216.11.141

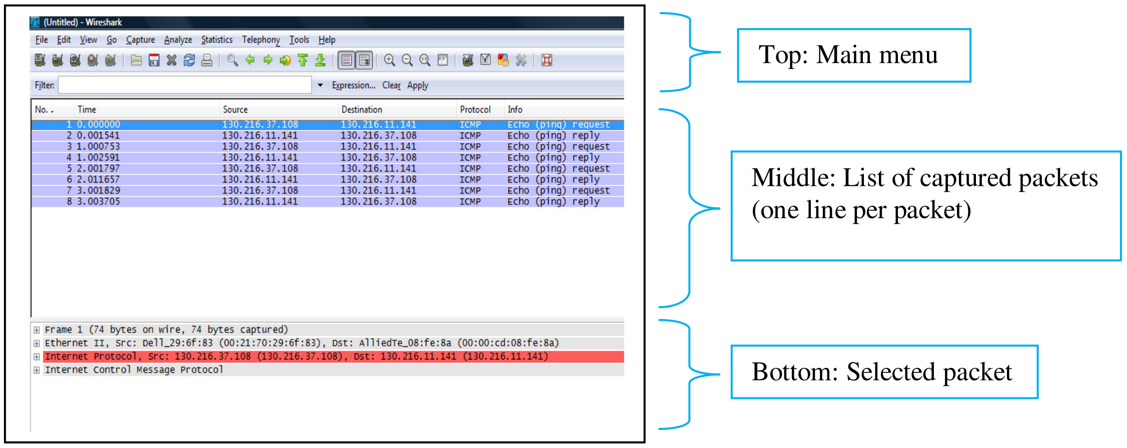
Capture File(s)
File:
[ ] Use multiple files
[✓] Next file every 1 megabyte(s)
[ ] Next file every 1 minute(s)
[✓] Ring buffer with 2 files
[ ] Stop capture after 1 file(s)

Stop Capture ...
[ ] ... after 1 packet(s)
[ ] ... after 1 megabyte(s)
[ ] ... after 1 minute(s)

Display Options
[✓] Update list of packets in real time
[✓] Automatic scrolling in live capture
[✓] Hide capture info dialog

Name Resolution
[✓] Enable MAC name resolution
[ ] Enable network name resolution
[✓] Enable transport name resolution

Figure 3. A screenshot of **Capture Options**

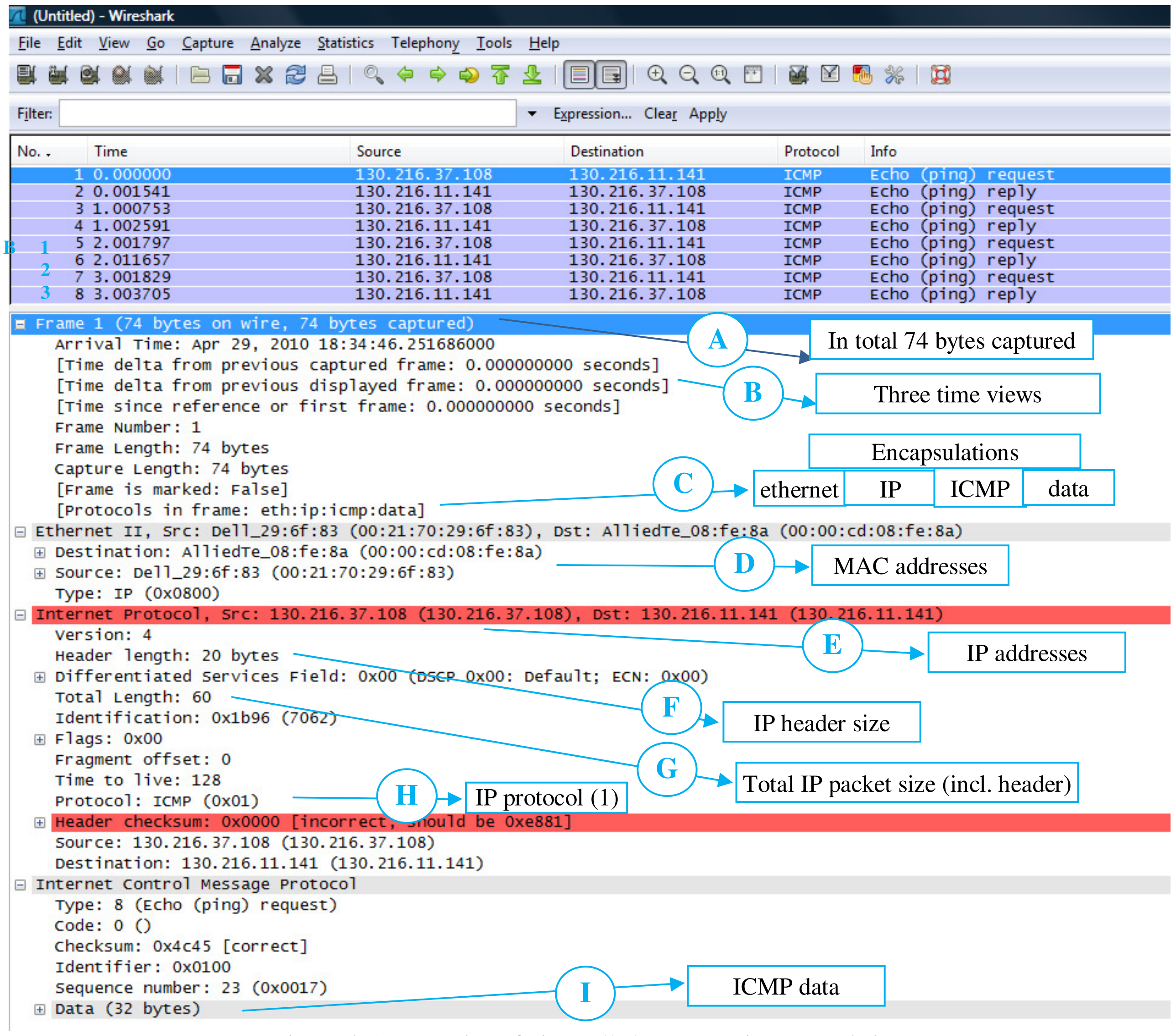

(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression... Clear Apply
Top: Main menu
Middle: List of captured packets (one line per packet)
Bottom: Selected packet

```
No. .  Time       Source           Destination      Protocol  Info
    1  0.000000   130.216.37.108   130.216.11.141   ICMP      Echo (ping) request
    2  0.001541   130.216.11.141   130.216.37.108   ICMP      Echo (ping) reply
    3  1.000753   130.216.37.108   130.216.11.141   ICMP      Echo (ping) request
    4  1.002591   130.216.11.141   130.216.37.108   ICMP      Echo (ping) reply
    5  2.001797   130.216.37.108   130.216.11.141   ICMP      Echo (ping) request
    6  2.011657   130.216.11.141   130.216.37.108   ICMP      Echo (ping) reply
    7  3.001829   130.216.37.108   130.216.11.141   ICMP      Echo (ping) request
    8  3.003705   130.216.11.141   130.216.37.108   ICMP      Echo (ping) reply
```

```
⊞ Frame 1 (74 bytes on wire, 74 bytes captured)
⊞ Ethernet II, Src: Dell_29:6f:83 (00:21:70:29:6f:83), Dst: AlliedTe_08:fe:8a (00:00:cd:08:fe:8a)
⊞ Internet Protocol, Src: 130.216.37.108 (130.216.37.108), Dst: 130.216.11.141 (130.216.11.141)
⊞ Internet Control Message Protocol
```

Figure 4. A screenshot of main screen showing three sections

You need to spend some time studying the middle and bottom sections of Figure 4. The middle section shows a list of captured packets in ascending time order. That is, the latest captured packet will be displayed at the bottom. You can however change to sort the packets in various orders, e.g., protocol by selecting the **Protocol** meta-header. You can select each packet, and the bottom layout will display detailed information about it. This is very powerful but can be complicated. We will explain the first captured packet (i.e., ICMP request packet). Here, you will see four lines (of details) at the bottom. You need to expand these by clicking plus [+] as shown in Figure 5.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression... Clear Apply
A
In total 74 bytes captured
B
Three time views
Encapsulations
C
ethernet
IP
ICMP
data
D
MAC addresses
E
IP addresses
F
IP header size
G
Total IP packet size (incl. header)
H
IP protocol (1)
I
ICMP data

```
No. .  Time        Source          Destination     Protocol  Info
    1  0.000000    130.216.37.108  130.216.11.141  ICMP      Echo (ping) request
    2  0.001541    130.216.11.141  130.216.37.108  ICMP      Echo (ping) reply
    3  1.000753    130.216.37.108  130.216.11.141  ICMP      Echo (ping) request
    4  1.002591    130.216.11.141  130.216.37.108  ICMP      Echo (ping) reply
    5  2.001797    130.216.37.108  130.216.11.141  ICMP      Echo (ping) request
    6  2.011657    130.216.11.141  130.216.37.108  ICMP      Echo (ping) reply
    7  3.001829    130.216.37.108  130.216.11.141  ICMP      Echo (ping) request
    8  3.003705    130.216.11.141  130.216.37.108  ICMP      Echo (ping) reply
```

```
⊟ Frame 1 (74 bytes on wire, 74 bytes captured)
    Arrival Time: Apr 29, 2010 18:34:46.251686000
    [Time delta from previous captured frame: 0.000000000 seconds]
    [Time delta from previous displayed frame: 0.000000000 seconds]
    [Time since reference or first frame: 0.000000000 seconds]
    Frame Number: 1
    Frame Length: 74 bytes
    Capture Length: 74 bytes
    [Frame is marked: False]
    [Protocols in frame: eth:ip:icmp:data]
⊟ Ethernet II, Src: Dell_29:6f:83 (00:21:70:29:6f:83), Dst: AlliedTe_08:fe:8a (00:00:cd:08:fe:8a)
  ⊞ Destination: AlliedTe_08:fe:8a (00:00:cd:08:fe:8a)
  ⊞ Source: Dell_29:6f:83 (00:21:70:29:6f:83)
    Type: IP (0x0800)
⊟ Internet Protocol, Src: 130.216.37.108 (130.216.37.108), Dst: 130.216.11.141 (130.216.11.141)
    Version: 4
    Header length: 20 bytes
  ⊞ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 60
    Identification: 0x1b96 (7062)
  ⊞ Flags: 0x00
    Fragment offset: 0
    Time to live: 128
    Protocol: ICMP (0x01)
  ⊞ Header checksum: 0x0000 [incorrect, should be 0xe881]
    Source: 130.216.37.108 (130.216.37.108)
    Destination: 130.216.11.141 (130.216.11.141)
⊟ Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0 ()
    Checksum: 0x4c45 [correct]
    Identifier: 0x0100
    Sequence number: 23 (0x0017)
  ⊞ Data (32 bytes)
```

Figure 5. A screenshot of Figure 4's bottom section *expanded*

A – Total of 74 bytes of frames is captured. 74 bytes are displayed for **on wire** and **captured**. This is because (by default) we've set to capture *full* (Figure 3). If we had set to limit the size to 68, we will see 74 bytes **on wire** and 68 bytes **captured**. Regardless of how small we set to limit the frame size, the NIC already finds **on wire** frame sizes (i.e., 74 bytes). Note that we are actually missing 4 bytes (of FCS) for each frame. This is because FCSs are already truncated by the NIC before *pcap* acquires the information. In other words, your frame size was actually 78 bytes.

B – Three timing lines are displayed. The first listed time (1) shows the time difference (*delta*) between the last captured frame and the currently selected frame. The second listed time (2) is similar to the first one, but calculates the time of last captured frame as what's displayed on the GUI (i.e., time since last frame selected by the filter expression in the main screen). The third listed time (3) shows the time of the currently selected captured frame *since* the capturing began.

The first line is useful to observe packet inter-arrival times (or packet gap times). Also, you will find the third line useful, as you can think of this as 'elapsed time since the packet capturing began'.

C – Here, you can see that the structure of a frame is **eth:ip:icmp:data**. This means that the first 14 bytes are the frame header, and the remaining 60 bytes are regarded as 'payload' of the frame.

D – Both source and destination frame addresses are shown. These are the NIC (MAC) addresses (Notice that they are 6 bytes each). We can further observe that the first 3 bytes is the manufacturer's ID and the other 3 bytes are the ID of the devices. For example, we observe that destination MAC address is AlliedTed_08:fe:8a (00:00:cd:08:fe:8a), most likely indicating that your ping command frames are traveling to a router/switch manufactured by *Allied Telesis*. You can disable MAC name resolution by unselecting 'Enable MAC name resolution' in the **Capture Options**.

E – From here, we are at IP packet level and similar to MAC addresses, both source and destination addresses are shown (they are 4 bytes each). Because we did not check 'Enable IP name resolution' in the **Capture Options**, no name lookups are performed. (i.e., this is the same as the **–n** option in windump).

F – This 20 bytes length is the IP header size and is the *minimum* size. Maximum possible IP header size is 60 bytes, e.g., additional 40 bytes of *option* field.

G – This 60 bytes length is the total IP packet size which includes the packet header. It therefore leaves 40 bytes of IP payload.

H – This is a protocol field in the IP header. It specifies what kind of application data the IP packet is carrying in its payload. The three most common protocols (ICMP, TCP, UDP) are 1, 6, and 17 respectively. Note that an IP packet can carry another IP packet, such as IPv4 packet carrying IPv6 packet. The protocol field would then be 41.
(refer link here, http://www.iana.org/assignments/protocol-numbers)

I – This 32 bytes is the payload of ICMP packet. We do not need to worry about the content of data here. Payload size implies the ICMP header size of 8 bytes (calculated from the remaining 40 bytes of IP packet payload).

Other than displaying packet information, there are more useful features to analyze the packets. A *filter* toolbar sitting on your main screen is use to filter out and display the packets you are interested in. For example, typing **tcp** and **apply** will only display packets that are TCP (Figure 6). There are more commands that you may find useful; refer examples by clicking on **Filter**.

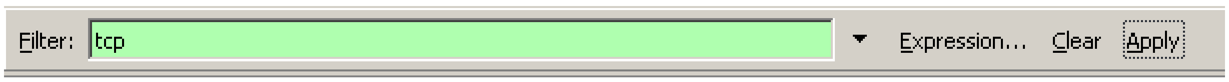
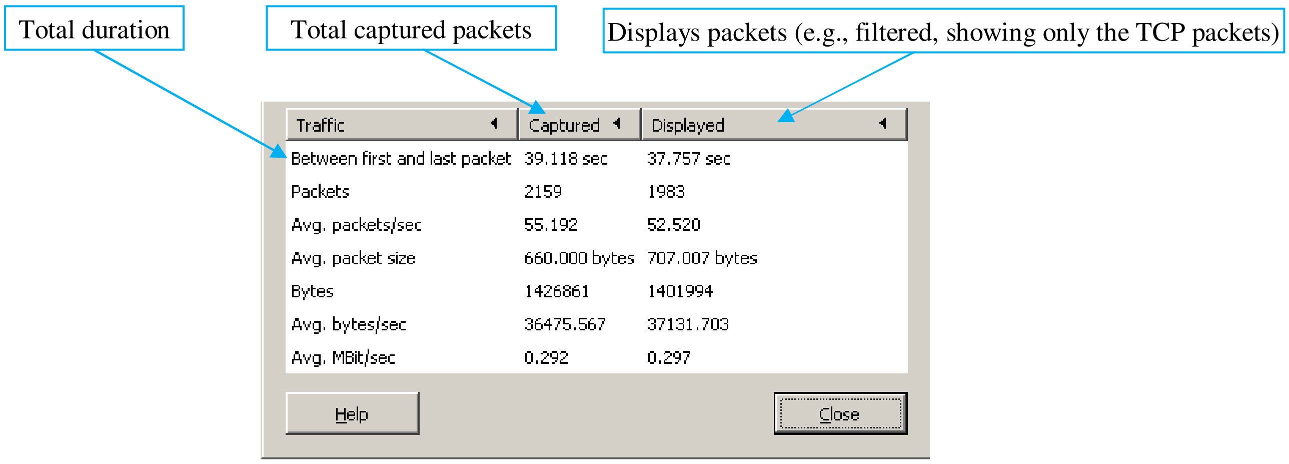


Figure 6. A filter toolbar

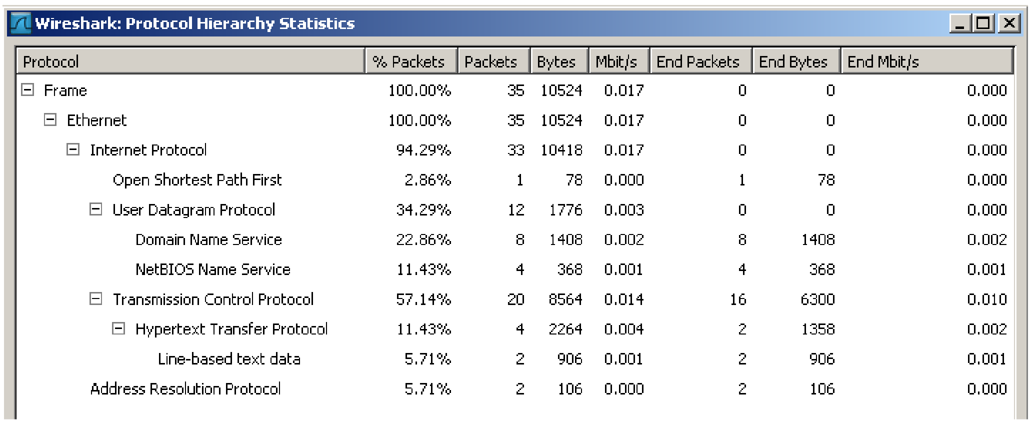
Summary under the **Statistics** menu gives overall traffic information. Figure 7 shows a summary of Captured and Displayed. Note that bytes are the total packet size **on wire**. Average rates of packets or bytes are calculated from the total counts over the duration.


Total duration
Total captured packets
Displays packets (e.g., filtered, showing only the TCP packets)
Help
Close

Traffic	Captured	Displayed
Between first and last packet	39.118 sec	37.757 sec
Packets	2159	1983
Avg. packets/sec	55.192	52.520
Avg. packet size	660.000 bytes	707.007 bytes
Bytes	1426861	1401994
Avg. bytes/sec	36475.567	37131.703
Avg. MBit/sec	0.292	0.297

Figure 7. Summary statistics

Protocol Hierarchy under the **Statistics** menu gives a protocol breakdown view. As shown in Figure 8, IP packets fall into two parts: UDP and TCP. Again these protocols are further separated (e.g., DNS and NetBIOS are the UDP packets).



Protocol	% Packets	Packets	Bytes	Mbit/s	End Packets	End Bytes	End Mbit/s
⊟ Frame	100.00%	35	10524	0.017	0	0	0.000
⊟ Ethernet	100.00%	35	10524	0.017	0	0	0.000
⊟ Internet Protocol	94.29%	33	10418	0.017	0	0	0.000
Open Shortest Path First	2.86%	1	78	0.000	1	78	0.000
⊟ User Datagram Protocol	34.29%	12	1776	0.003	0	0	0.000
Domain Name Service	22.86%	8	1408	0.002	8	1408	0.002
NetBIOS Name Service	11.43%	4	368	0.001	4	368	0.001
⊟ Transmission Control Protocol	57.14%	20	8564	0.014	16	6300	0.010
⊟ Hypertext Transfer Protocol	11.43%	4	2264	0.004	2	1358	0.002
Line-based text data	5.71%	2	906	0.001	2	906	0.001
Address Resolution Protocol	5.71%	2	106	0.000	2	106	0.000

Figure 8. Protocol Hierarchy statistics

According to experts, the Internet will face a serious problem in near future. Internet Protocol Version 4 (IPv4) is the native protocol of the Internet at the moment. As you have seen in the examples, IPv4 addresses are 32 bits wide, so they are able to address 2^{32} hosts all around the world. As Internet is growing repidly, no more free IPv4 addresses will be available in near future (An estimate is 2012). Several solutions have been proposed for solving this problem. Switching to a new protocol that does not have IPv4 problems is one of them. Version 6 of the Internet Protocol has been designed to fulfill Internet demand for a bigger address space, although it's not the only advantage of IPv6. It offers some features which make it more secure and also more suitable for mobility.

IPv6 offers a huge address space by employing 128 bits addresses which means 667×10^{21} addresses for each square meter of the earth or 5×10^{28} addresses for each of people alive today. With this amount of addresses, there is no need to use troublesome techniques like NAT. IPv6 is not widely deployed but there are some sites, like ipv6.google.com, which have that enabled.

Wireshark is able to capture and decode IPv6 packets. Figure 9 shows a screen shot of Wireshark with some captured IPv6 packets. These packets are ICMP packets which were generated by the following ping command.

```
D:\> Ping ipv6.google.com

Pinging ipv6.l.google.com [2404:6800:8004::68] from 2001:df0:0:2005:b12b:1e0:f3c1:a721 with 32
bytes of data:
Reply from 2404:6800:8004::68: time=32ms
Reply from 2404:6800:8004::68: time=27ms
Reply from 2404:6800:8004::68: time=26ms
Reply from 2404:6800:8004::68: time=31ms

Ping statistics for 2404:6800:8004::68:
        Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
        Minimum = 26ms, Maximum = 32ms, Average = 29ms
```

Look at figures 5 and 9. IPv6 header has fewer fields than IPv4. IPv6 designers have tried to make the IP header as small and simple as possible to accelerate IP header processing in network hosts and routers. IPv6 header only contains basic fields. Packets which need to carry more data in their IP header will need to use extension headers. Figure 10 shows an example of an ICMP packet which carries an extension header.

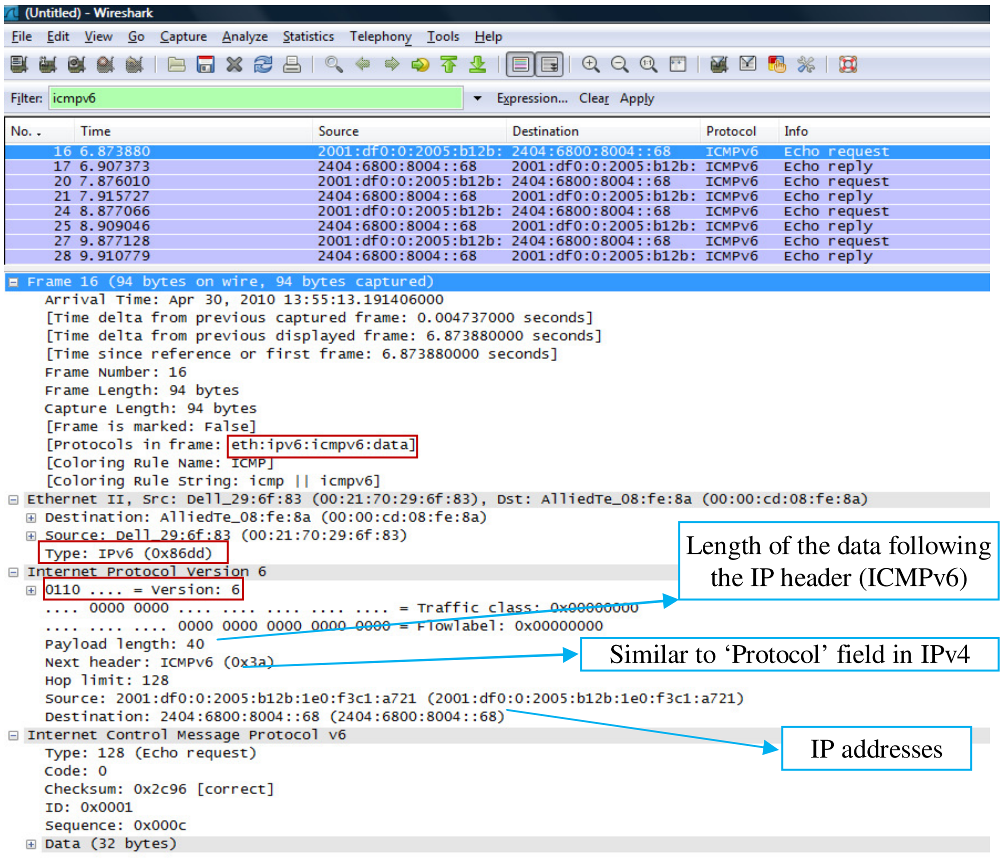
To represent IPV6 addresses, hexadecimal notation is used: 8 fields of hexadecimal numbers.
Following rules should also be considered:

- letters are case sensitive
- leading zeros in a field are optional: 00c1 = c1
- successive fields of '0' are represented as ::, **but just once** in an address. So, an address like 2001 _ 0000 _ 1234 _ **0000** _ **0000** _ C1C0 _ ABCD _ 0876 can be represented as 2001:0:1234**::**C1C0:ABCD:876.

You can use nslookup to find IPv6 addresses. By default, nslookup only shows IPv4 addresses. Use *set type=AAAA* command for retrieving IPv6 addresses:

```
D:\>nslookup www.auckland.ac.nz
Default Server:  kronos1.cs.auckland.ac.nz
Address:  130.216.35.35:35
```

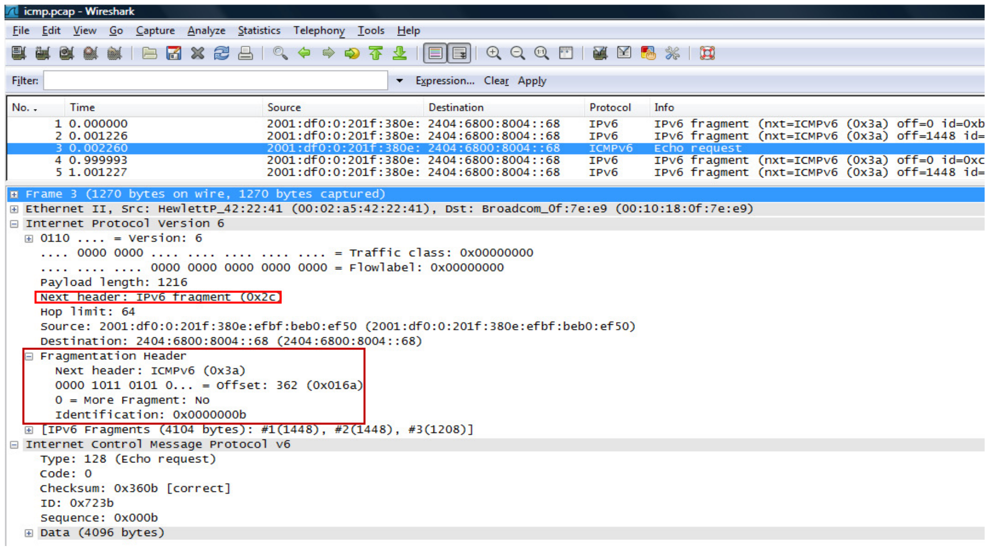
```
>set type=AAAA
>ipv6.google.com
Name:    ipv6.l.google.com
Address:  2404:6800:8004::68
Aliases:  ipv6.google.com
```


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter: icmpv6
Expression... Clear Apply
Length of the data following the IP header (ICMPv6)
Similar to 'Protocol' field in IPv4
IP addresses

```
No. .  Time          Source                    Destination               Protocol  Info
   16 6.873880     2001:df0:0:2005:b12b:   2404:6800:8004::68        ICMPv6    Echo request
   17 6.907373     2404:6800:8004::68        2001:df0:0:2005:b12b:   ICMPv6    Echo reply
   20 7.876010     2001:df0:0:2005:b12b:   2404:6800:8004::68        ICMPv6    Echo request
   21 7.915727     2404:6800:8004::68        2001:df0:0:2005:b12b:   ICMPv6    Echo reply
   24 8.877066     2001:df0:0:2005:b12b:   2404:6800:8004::68        ICMPv6    Echo request
   25 8.909046     2404:6800:8004::68        2001:df0:0:2005:b12b:   ICMPv6    Echo reply
   27 9.877128     2001:df0:0:2005:b12b:   2404:6800:8004::68        ICMPv6    Echo request
   28 9.910779     2404:6800:8004::68        2001:df0:0:2005:b12b:   ICMPv6    Echo reply

⊟ Frame 16 (94 bytes on wire, 94 bytes captured)
    Arrival Time: Apr 30, 2010 13:55:13.191406000
    [Time delta from previous captured frame: 0.004737000 seconds]
    [Time delta from previous displayed frame: 6.873880000 seconds]
    [Time since reference or first frame: 6.873880000 seconds]
    Frame Number: 16
    Frame Length: 94 bytes
    Capture Length: 94 bytes
    [Frame is marked: False]
    [Protocols in frame: eth:ipv6:icmpv6:data]
    [Coloring Rule Name: ICMP]
    [Coloring Rule String: icmp || icmpv6]
⊟ Ethernet II, Src: Dell_29:6f:83 (00:21:70:29:6f:83), Dst: AlliedTe_08:fe:8a (00:00:cd:08:fe:8a)
  ⊞ Destination: AlliedTe_08:fe:8a (00:00:cd:08:fe:8a)
  ⊞ Source: Dell_29:6f:83 (00:21:70:29:6f:83)
    Type: IPv6 (0x86dd)
⊟ Internet Protocol Version 6
  ⊞ 0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload length: 40
    Next header: ICMPv6 (0x3a)
    Hop limit: 128
    Source: 2001:df0:0:2005:b12b:1e0:f3c1:a721 (2001:df0:0:2005:b12b:1e0:f3c1:a721)
    Destination: 2404:6800:8004::68 (2404:6800:8004::68)
⊟ Internet Control Message Protocol v6
    Type: 128 (Echo request)
    Code: 0
    Checksum: 0x2c96 [correct]
    ID: 0x0001
    Sequence: 0x000c
  ⊞ Data (32 bytes)
```

Figure 9. A screenshot of Wireshark with some IPv6 packets


icmp.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression... Clear Apply

```
No. .  Time        Source                    Destination           Protocol  Info
    1 0.000000   2001:df0:0:201f:380e:   2404:6800:8004::68    IPv6      IPv6 fragment (nxt=ICMPv6 (0x3a) off=0 id=0xb
    2 0.001226   2001:df0:0:201f:380e:   2404:6800:8004::68    IPv6      IPv6 fragment (nxt=ICMPv6 (0x3a) off=1448 id=
    3 0.002260   2001:df0:0:201f:380e:   2404:6800:8004::68    ICMPv6    Echo request
    4 0.999993   2001:df0:0:201f:380e:   2404:6800:8004::68    IPv6      IPv6 fragment (nxt=ICMPv6 (0x3a) off=0 id=0xc
    5 1.001227   2001:df0:0:201f:380e:   2404:6800:8004::68    IPv6      IPv6 fragment (nxt=ICMPv6 (0x3a) off=1448 id=

⊞ Frame 3 (1270 bytes on wire, 1270 bytes captured)
⊞ Ethernet II, Src: HewlettP_42:22:41 (00:02:a5:42:22:41), Dst: Broadcom_0f:7e:e9 (00:10:18:0f:7e:e9)
⊟ Internet Protocol Version 6
  ⊞ 0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00000000
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
    Payload length: 1216
    Next header: IPv6 fragment (0x2c)
    Hop limit: 64
    Source: 2001:df0:0:201f:380e:efbf:beb0:ef50 (2001:df0:0:201f:380e:efbf:beb0:ef50)
    Destination: 2404:6800:8004::68 (2404:6800:8004::68)
  ⊟ Fragmentation Header
      Next header: ICMPv6 (0x3a)
      0000 1011 0101 0... = Offset: 362 (0x016a)
      0 = More Fragment: No
      Identification: 0x0000000b
  ⊞ [IPv6 Fragments (4104 bytes): #1(1448), #2(1448), #3(1208)]
⊟ Internet Control Message Protocol v6
    Type: 128 (Echo request)
    Code: 0
    Checksum: 0x360b [correct]
    ID: 0x723b
    Sequence: 0x000b
  ⊞ Data (4096 bytes)
```

Figure 10. An ICMPv6 packet with fragmentation header

VIII. *Optional Programming* – Capturing the packets in Java

A. *Jpcap: Java wrapper for pcap*

This section is optional, where you will learn to program in Java to capture and send packets using *jpcap* library. Although Wireshark has many useful functions that can be extended, there are some limitations to what you can do, for instance, you cannot customize or aggregate packets. The main function of this jpcap library is the ability to call its APIs under the Java environment. This means that once you've learned some of the APIs, you can capture the packets and process them on your own.
Jpcap main page: http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html
Jpcap API page: http://netresearch.ics.uci.edu/kfujii/jpcap/doc/javadoc/index.html

Rather than explaining each API or method, we will briefly start with a simple tool provided by the jpcap author. Figure 11 is a simple Java code (*Tcpdump.java*). You should be able to understand it as most of the codes are self-explanatory. The program works almost the same as how windump prints out the captured packets.

```
import jpcap.*;
import jpcap.packet.Packet;

class Tcpdump implements PacketReceiver {

        public void receivePacket(Packet packet) {          C
                System.out.println(packet);
        }

        public static void main(String[] args) throws Exception {
                NetworkInterface[] devices = JpcapCaptor.getDeviceList();
                if(args.length<1){
                        System.out.println("usage: java Tcpdump <select a number from the following>");
                        for (int i = 0; i < devices.length; i++) {
                                System.out.println(i+" :"+devices[i].name + "(" + devices[i].description+")");
                                System.out.println("    data link:"+devices[i].datalink_name + "("+ devices[i].datalink_description+")");
                   A            System.out.print("    MAC address:");
                                for (byte b : devices[i].mac_address)
                                        System.out.print(Integer.toHexString(b&0xff) + ":");
                                System.out.println();
                                for (NetworkInterfaceAddress a : devices[i].addresses)
                                        System.out.println("    address:"+a.address + " " + a.subnet + " "+ a.broadcast);
                        }
                }else{
                        JpcapCaptor jpcap = JpcapCaptor.openDevice(devices[Integer.parseInt(args[0])], 2000, false, 20);
                  B     jpcap.loopPacket(-1, new Tcpdump());
                }
        }
}
```

Figure 11. Simple Tcpdump.java code (http://netresearch.ics.uci.edu/kfujii/jpcap/sample/Tcpdump.java)

A – This part is the start of the program where the main method exists. If there are no arguments, then it will print out a list of network interfaces available by the pcap, and finishes the program. The output is similar to **windump –D** command. Once you've added a device number as an argument, it will jump to the *else* part.

B – Here, it creates a capturing handler instance, *jpcap* using the argument. Then it calls out a *loopPacket* method to capture packets observed for that instance (i.e., your NIC). Note that first parameter (-1) in *loopPacket* specifies to capture continuously until the program is aborted by user, and the second parameter calls out new instances of *Tcpdump* which (must) implement the *PacketReceiver* interface. Thus, every captured packet will call out an interface method *receivePacket*.

C – As mentioned, this method is called for each packet received in *loopPacket()*. In this method, it simply 'prints' out the *packet* instance, which acts similar to *toString()*. In other words, it will print out brief information of the packet, similar to the default output of windump. This *receivePacket()* is the method you should modify: processing the packets and building your own small monitoring tool. There are a few more examples in the above website. You should attempt to modify some of these to get used to programming and the jpcap API.

B. *Compiling and running the code*

Because you cannot install jpcap into the System or Java directory, you need to acquire two files ***jpcap.jar*** and ***jpcap.dll*** (located at the *Resource* section). For simplicity, copy both files into your directory where you will be working. To compile, type: `javac Tcpdump.java –classpath jpcap.jar`
You can run the program by typing `java Tcpdump`
If you prefer to use Textpad, you need to go to **Configure**, **Preferences**, **Tools** and **Compile Java**. Modify the **Parameters** to `–classpath jpcap.jar $File`
Note that you can do similar for other IDE tools.

You may run into the Exception *OutOfMemoryError* while your program is running: you need to allocate more heap size to the JVM, for example, `java –Xmx512M ...` will allocate a maximum amount of 512MB to it.

C. *Jpcap exercises*

You should build your own small monitoring tool using this library. Your program should produce some simple statistics of the captured packets, and provide useful features that cannot be done in Wireshark. We will provide some trace files for you to test on, located at the *Resource* section. Note that you should develop with care: capture a few packets, double check that what you are observing is also correct with Wireshark. You can assume that Wireshark shows the correct answers. In particular, here are some examples you could do:

- **Brief Statistics1**: Total packets, total TCP packets, total UDP packets, total ICMP packets, total non-IP packets, total duration of the trace (elapsed time), and average rates.
- **Brief Statistics2**: timestamps of each packet and packet inter-arrival times.
- **Filtering**: filtering number of known IP addresses, TCP/UDP port numbers.
- **Activity Log**: finding out some activities, e.g., alarming when it detects a large file transfer.