Networks

Table of Contents

Network theory as abstract topics is not easy. These notes look at some practical aspects of networking, to see at the byte level what actually happens.

This is done on an x86/64 desktop PC running Linux Debian Mint 18.04 on a home network.

Protocols and code

TCP/IP is a set of protocols, organised into layers, with each layer doing a broadly similar task - for example link layer protocols are about communication between devices on the same network.

A protocol is a set of rules - a specification. Protocols are usually set out in an RFC from the IETF.

Code to implement a protocol is often part of an operating system kernel. The OS might use that code to use the protocol, without the user needing to know what is happening.

There might also be an API, so that those features can be called from a C program (or similar language).

There might also be an application, command-line or GUI, to monitor or control the protocol.

Linux command line utilities

There are a set of Linux command line utilities relevant to networks.

These can be used to see aspects of the local network and the Internet. They can also be used to configure these systems, and this requires care - or you might accidentally configure your system to prevent Internet access, and so be unable to find out how to unconfigure it.

There are a set of network uitilities in a package named net-tools, which is now considered obsolete. These include arp, ifconfig and route. The replacement is called iproute2. This does several functions through an application called ip, with different command line switches:

Some tools and purposes are:

Purpose	Command
Show interfaces on local host	ip addr
Show devices on local network	nmap
Show routing on local host	ip route
Show route to remote host	tracert
Time route to remote host	ping

Show interfaces

To display information about the network interfaces on the current host, we can say ip addr.

For example:

```
walter@mint2 ~ $ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
    link/ether 74:d4:35:54:a8:9f brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.65/24 brd 192.168.1.255 scope global dynamic enp1s0
       valid_lft 60510sec preferred_lft 60510sec
    inet6 fe80::b8ca:93e5:7d1a:258d/64 scope link
       valid_lft forever preferred_lft forever
3: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group
default qlen 1000
    link/ether 74:da:38:32:7b:fa brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.79/24 brd 192.168.1.255 scope global dynamic wlp2s0
       valid_lft 78317sec preferred_lft 78317sec
    inet6 fe80::c6ef:923c:6d3d:a6a9/64 scope link
       valid_lft forever preferred_lft forever
```

This shows there are 3 interfaces. The first has the name lo, and is the local loop - ip address 127.0.0.1. The second is enp1s0 and is an Ethernet twisted pair cable, connected to a router gateway, and the third is wlp2s0, a wireless interface.

The local loop has a mtu - maximum transmitted unit - of 65536, so packets cannot be bigger than 64k. qdisc, for queing discipline, is how queueing is handled. The state is UNKNOWN because this is reported by the device driver, and loopback does not have a driver. UP means the interface is switched on - that is, configured to be usable. LOWER_UP means whether the hardware is OK.

The IP address of this is given as 127.0.0.1/8. The 8 means the first 8 bits of this is for the LAN, and the rest are for hosts on the LAN. So the actual addresses are 127.0.0.1 to 127.255.255.255. For example

```
walter@mint2 ~ $ ping 127.34.56.77
PING 127.34.56.77 (127.34.56.77) 56(84) bytes of data.
64 bytes from 127.34.56.77: icmp_seq=1 ttl=64 time=0.059 ms
64 bytes from 127.34.56.77: icmp_seq=2 ttl=64 time=0.043 ms
64 bytes from 127.34.56.77: icmp_seq=3 ttl=64 time=0.049 ms
64 bytes from 127.34.56.77: icmp_seq=4 ttl=64 time=0.048 ms
64 bytes from 127.34.56.77: icmp_seq=5 ttl=64 time=0.048 ms
^C
--- 127.34.56.77 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3998ms
rtt min/avg/max/mdev = 0.043/0.049/0.059/0.008 ms
```

The Ethernet interface has a MAC address of 74:d4:35:54:a8:9f. If we unplug the cable and run ip addr again, we get:

```
2: enp1s0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state
DOWN group default qlen 1000
```

Show devices on LAN

This can be done with nmap, which is an open source cross-platform utility.

For example:

```
sudo nmap -sn 192.168.1.0/24

Starting Nmap 7.01 ( https://nmap.org ) at 2021-02-28 16:53 GMT
Nmap scan report for walter-s5-1030uk.lan (192.168.1.64)
Host is up (0.0045s latency).
MAC Address: E0:69:95:8D:A8:E4 (Pegatron)
Nmap scan report for DESKTOP-Q4M2RIN.lan (192.168.1.77)
Host is up (-0.094s latency).
MAC Address: C0:7C:D1:FD:D4:FD (Pegatron)
Nmap scan report for Honor_10-a2224b467bd06516.lan (192.168.1.82)
Host is up (0.32s latency).
MAC Address: C0:F4:E6:68:35:B5 (Unknown)
Nmap scan report for android-dhcp-9-d4-91-0f-40-8c-79.lan (192.168.1.86)
```

```
Host is up (-0.092s latency).
MAC Address: D4:91:0F:40:8C:79 (Unknown)
Nmap scan report for dsldevice.lan (192.168.1.254)
Host is up (0.0046s latency).
MAC Address: A0:1B:29:67:4E:CA (Sagemcom Broadband SAS)
Nmap scan report for mint2.lan (192.168.1.65)
Host is up.
Nmap scan report for mint2.lan (192.168.1.79)
Host is up.
Nmap done: 256 IP addresses (7 hosts up) scanned in 5.98 seconds
```

nmap -sn 192.168.1.0/24. The -sn means do not do a port scan of every device - which is slow. 192.168.1.0/24 means all the possible private IP addresses on a network of this type.

```
Nmap scan report for walter-s5-1030uk.lan (192.168.1.64)
Host is up (0.0045s latency).
MAC Address: E0:69:95:8D:A8:E4 (Pegatron)
```

walter-s5-1030uk.lan is the host name for a desktop PC running Ubuntu, with private IP address 192.168.1.64.
DESKTOP-Q4M2RIN.lan (192.168.1.77) is a Windows X desktop
Honor_10-a2224b467bd06516.lan (192.168.1.82) is an Android smartphone
android-dhcp-9-d4-91-0f-40-8c-79.lan (192.168.1.86) is a Chromebook
dsldevice.lan (192.168.1.254) is a router gateway
mint2.lan (192.168.1.65) is a desktop running Debian Mint connected by Ethernet
mint2.lan (192.168.1.79) is the same machine, but through a wireless interface

Traceroute

This is a utility which attempts to show which routers are used on the path to a remote host (tracert on Windows). It uses TCP or UDP packets, depending on the implementation. For example

```
walter@mint2 ~ $ traceroute --resolve-hostnames -q 1 example.com
traceroute to example.com (93.184.216.34), 64 hops max
  1   192.168.1.254 (dsldevice.lan)  1.138ms
  2   *
  3   *
  4   195.166.143.140 (140.hiper04.sheff.dial.plus.net.uk)  8.405ms
  5   195.99.125.138 (195.99.125.138)  8.760ms
  6   109.159.252.90 (peer7-et-0-1-5.telehouse.ukcore.bt.net)  8.544ms
  7   166.49.214.194 (166-49-214-194.gia.bt.net)  7.222ms
  8   166.49.195.61 (ixp1-ae-7.us-nyc.gia.bt.net)  74.303ms
  9   *
 10   152.195.69.131 (ae-66.core1.nyb.edgecastcdn.net)  75.782ms
```

```
 11   93.184.216.34 (93.184.216.34)  76.061ms
 12   93.184.216.34 (93.184.216.34)  75.581ms
```

The -q 1 command line option means try once - not the default 3 times.
--resolve-hostnames means display IP addresses and their corresponding domain names. The router may not respond, in which case we get a * displayed.

In this example,
1 is my router-gateway, at private address 192.168.1.254
4 and 5 are PlusNet routers, my ISP
7 is a BT router, and 8 is BT in New York,

We can get more details using whois:

```
walter@mint2 ~ $ whois 152.195.69.131
..
NetRange:       152.176.0.0 - 152.199.255.255
CIDR:           152.192.0.0/13, 152.176.0.0/12
NetName:        UU-152-176
NetHandle:      NET-152-176-0-0-1
Parent:         NET152 (NET-152-0-0-0-0)
NetType:        Direct Allocation
OriginAS:       AS1321, AS701
Organization:   ANS Communications, Inc (ANS)
RegDate:        1992-04-01
Updated:        2016-08-18
Comment:        Addresses within this block are non-portable.
Ref:            https://rdap.arin.net/registry/ip/152.176.0.0

OrgName:        ANS Communications, Inc
OrgId:          ANS
Address:        22001 Loudoun County Parkway
City:           Ashburn
StateProv:      VA
PostalCode:     20147
Country:        US
RegDate:        1991-07-12
Updated:        2009-12-07
Ref:            https://rdap.arin.net/registry/entity/ANS

OrgTechHandle: SWIPP-ARIN
OrgTechName:   swipper
OrgTechPhone:  +1-800-900-0241
OrgTechEmail:  swipper@verizonbusiness.com
OrgTechRef:    https://rdap.arin.net/registry/entity/SWIPP-ARIN

OrgNOCHandle: OA12-ARIN
OrgNOCName:   UUnet Technologies, Inc., Technologies
OrgNOCPhone:  +1-800-900-0241
OrgNOCEmail:  help4u@verizonbusiness.com
OrgNOCRef:    https://rdap.arin.net/registry/entity/OA12-ARIN
..
```

PING

This utility uses ICMP packets to test the reachability of a remote host. For example:

```
walter@mint2 ~ $ ping -c 5 example.com
PING example.com (93.184.216.34) 56(84) bytes of data.
64 bytes from 93.184.216.34: icmp_seq=1 ttl=53 time=75.5 ms
64 bytes from 93.184.216.34: icmp_seq=2 ttl=53 time=75.9 ms
64 bytes from 93.184.216.34: icmp_seq=3 ttl=53 time=75.4 ms
64 bytes from 93.184.216.34: icmp_seq=4 ttl=53 time=75.5 ms
64 bytes from 93.184.216.34: icmp_seq=5 ttl=53 time=75.3 ms

--- example.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 75.307/75.560/75.921/0.268 ms
```

The ttl is the 'time to live' - a count of the hops needed. The inital value of this can be set with the -t option. For example:

```
walter@mint2 ~ $ ping -c 5 -t 10 example.com
PING example.com (93.184.216.34) 56(84) bytes of data.
From ae-66.core1.nyb.edgecastcdn.net (152.195.69.131) icmp_seq=1 Time to live
exceeded
From ae-71.core1.nyb.edgecastcdn.net (152.195.69.139) icmp_seq=2 Time to live
exceeded
From ae-71.core1.nyb.edgecastcdn.net (152.195.69.139) icmp_seq=3 Time to live
exceeded
From ae-71.core1.nyb.edgecastcdn.net (152.195.69.139) icmp_seq=4 Time to live
exceeded
From ae-71.core1.nyb.edgecastcdn.net (152.195.69.139) icmp_seq=5 Time to live
exceeded

--- example.com ping statistics ---
5 packets transmitted, 0 received, +5 errors, 100% packet loss, time 4004ms
```

The ttl was initially 10. This got as far as 152.195.9.139, where it was reduced to 0, and the packet was eaten.

Telnet and SSH

Telnet is an application layer protocol intended to allow a user on one host to connect to a remote host, login, and issue commands typed on the local console, with the commands executed on the remote host, and output on the remote host displayed on the local host.

It is connection-oriented and uses TCP.

However telnet does not encrypt the data sent, including passwords, so it is not usually used.

Instead people use ssh, secure shell. This is in effect the same, but data is encrypted.

ssh is usually distributed with Linux, but may be disabled. It can be enabled and started by

```
sudo systemctl enable ssh
sudo systemctl start ssh
```

Then from another machine:

```
walter@mint2 ~ $ ssh 192.168.1.87 -l pi
pi@192.168.1.87's password:
Linux raspberrypi 5.10.17+ #1403 Mon Feb 22 11:26:13 GMT 2021 armv6l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Apr  4 10:37:08 2021 from 192.168.1.65
pi@raspberrypi:~ $ ls
Bookshelf  Desktop  Documents  Downloads  Music  Pictures  Public  Templates
Videos
pi@raspberrypi:~ $ exit
logout
Connection to 192.168.1.87 closed.
walter@mint2 ~ $
```

Here 192.168.1.87 is the IP address of the Raspberry Pi on the same LAN. -l pi sets the login name on it.

Using Wireshark

Wireshark

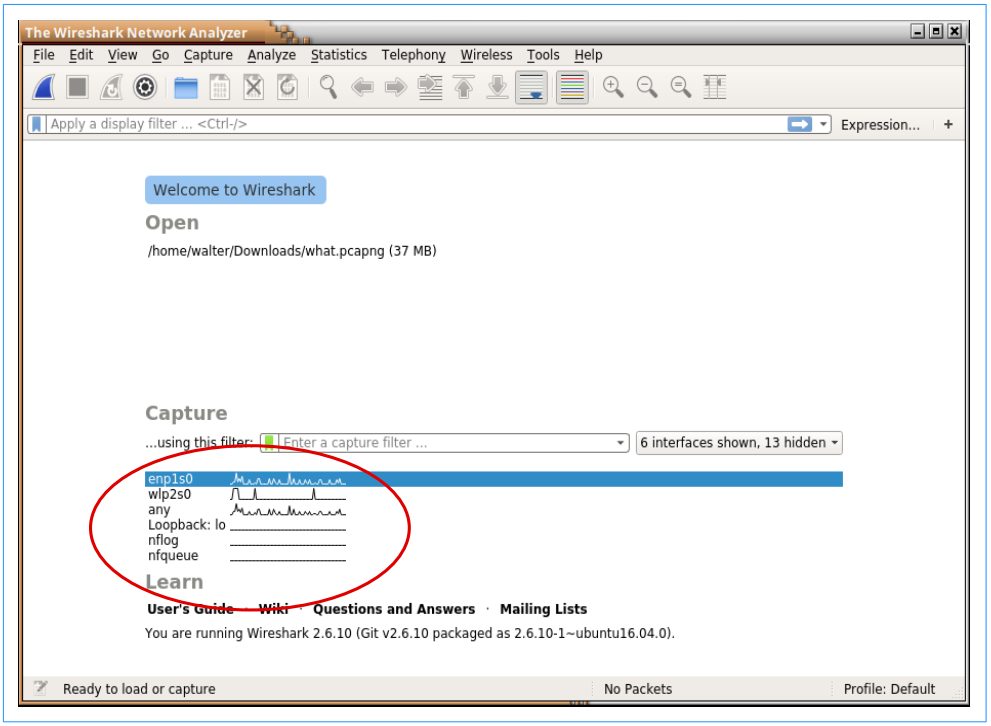
Wireshark is a GUI application for Linux and Windows which allows us to look at packets on a network. It was formerly called Ethereal. It is a free download simple to install.

It may need root priveledges to run - so start it by

```
sudo wireshark
```

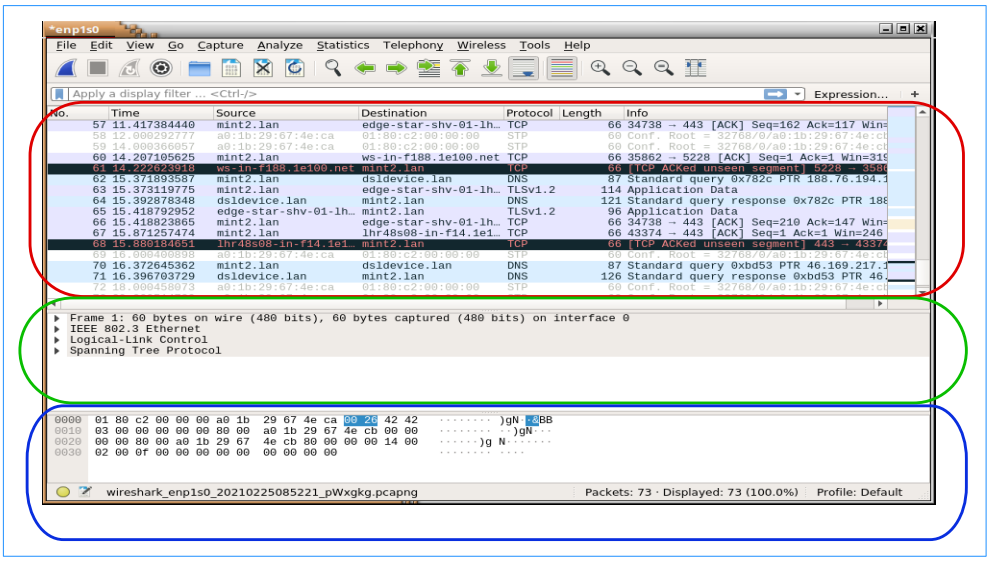
Screenshots are for version 2.6.10

The initial screen looks like:



It will display a list of the network interfaces on the device. enps0 is a wired Ethernet connection, wlp2s0 is a wireless connection, and Loopbacklo is the local loop. For each interface we see a small graph against time of the number of packets being seen.

Double-click an interface to start capturing the packets on it. See a stream of packets like this. Click the red square top left to stop the capture. The blue shark fin starts it.



The main window has 3 areas. The top red part shows the packets, with one row per packet.

The second green area shows the parts of the one packet selected. These parts depend on what protocol the packet is. Click the arrow to expand the fields.

The blue bottom area shows the actual bytes in the frame. To the left are the bytes, in hex. To the right are the bytes as ASCII characters - which will only make sense if the bytes are text data.

NTP

NTP is net time protocol. Its purpose is to enable computers on the net to synchronise their clocks. We use this as an example because it is simple.

The idea is that there are a small number of very accurate clocks, some of them atomic. These are called stratum 0 servers. Other machines synchronise themselves from these, and with each other - they are called stratum 1. Stratum 2 servers sync with stratum 1. And so on.

Any host on the Internet can get a time check from a time server on stratum 1 and below. This is not a simple process, because of network latency - the time delay between a signal sent from a server, and being received.

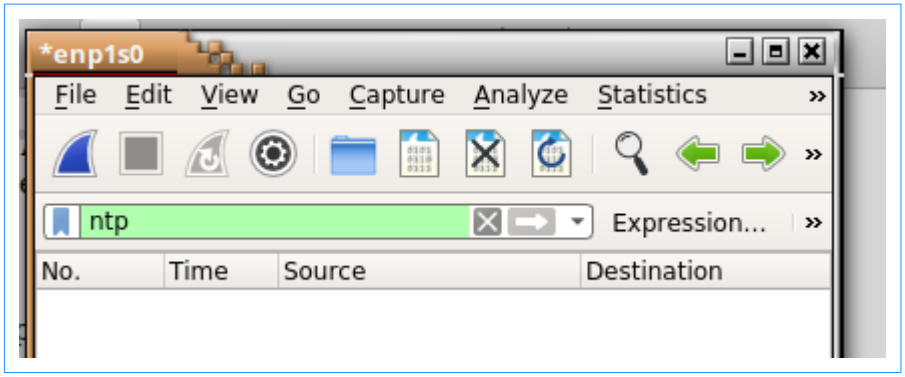
Linux has a pair of programs about this. chronyd is a daemon - a process which the OS runs in the background. chronyc is a command line program to interact with the daemon.

Check you have it installed by typing..

```
chronyc
chrony version 2.1.1
Copyright (C) 1997-2003, 2007, 2009-2015 Richard P. Curnow and others
chrony comes with ABSOLUTELY NO WARRANTY.  This is free software, and
you are welcome to redistribute it under certain conditions.  See the
GNU General Public License version 2 for details.
```

Otherwise follow the instructions to install it.

Then set up WireShark to filter only ntp packets:



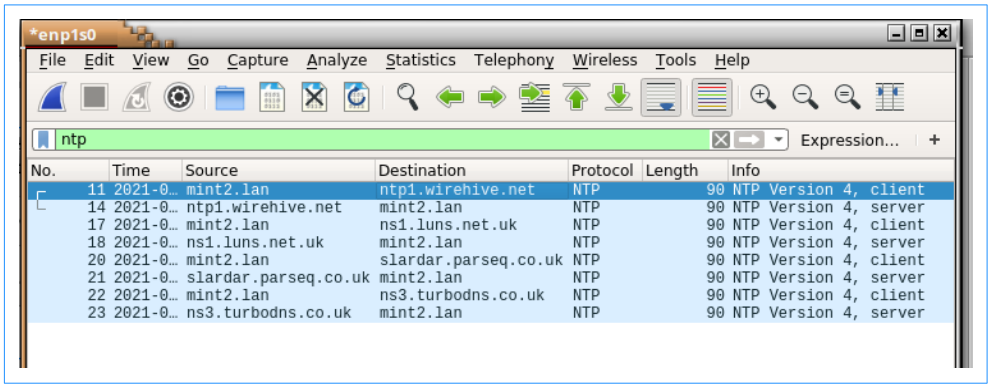
and click the blue fin to start capturing.

We want to force chrony to do a time check. Type:

```
sudo chronyc -a 'burst 1/1'
[sudo] password for walter:
200 OK
200 OK
```

This makes chrony do a time check on its default time servers. The 1/1 means check each server at least 1 time and at most 1 time.

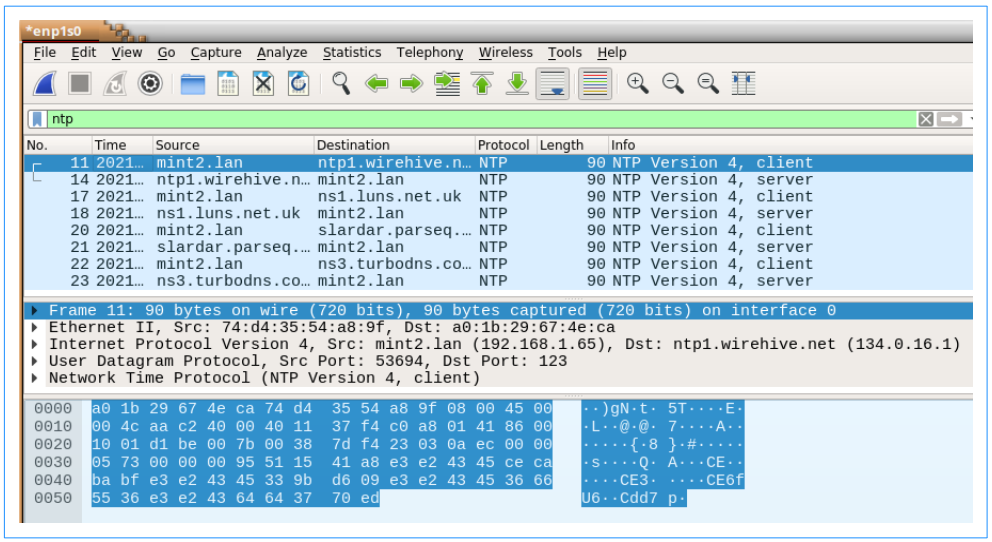
WireShark shows something like:



This shows 8 packets. chrony is using 4 default time servers - ntp1.wirehive.net, ns1.luns.net.uk, slardar.parseq.co.uk, and ns3.turbodns.co.uk. This machine is mint2.lan. It has sent a time request to each server, and received a packet back from each. WireShark shows a [over related packets - the first two, for example.

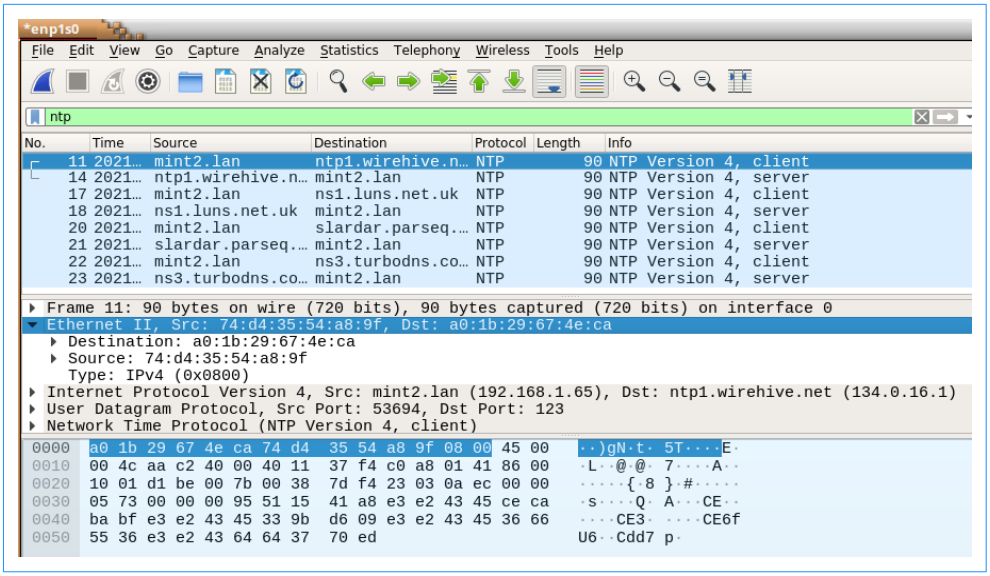
The packets are sent by UDP. This is connectionless - packets are just sent, and they may arrive and get a reply, or not. Here we have been lucky - all 4 have replied.

If we select one packet, and the frame section, we see:



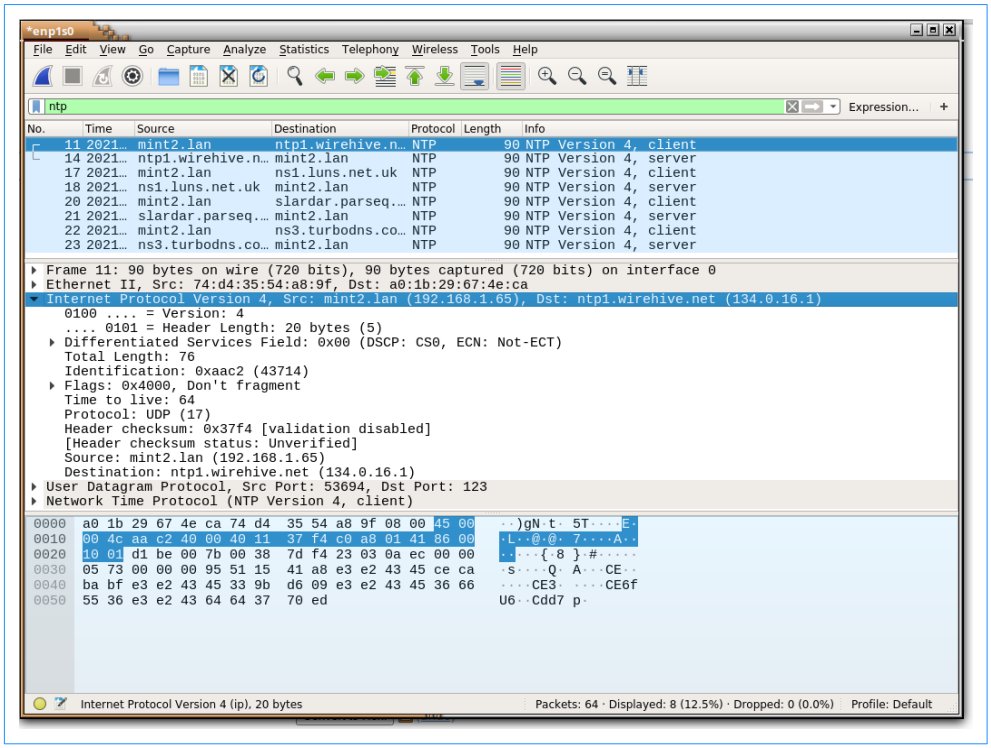
This is the entire frame, 90 bytes in length, captured off the Ethernet interface. It is frame 11. Frames 1 to 10 were not NTP and so were not filtered.

If we click the Ethernet II, we see:



This shows the bytes added by the link layer protocol. There are just 3 fields - the MAC address of the source (this machine 74:d4:35..) the destination (the router, a0:1b..) and the type, IPv4. These 14 bytes are highlighted in the bottom section.

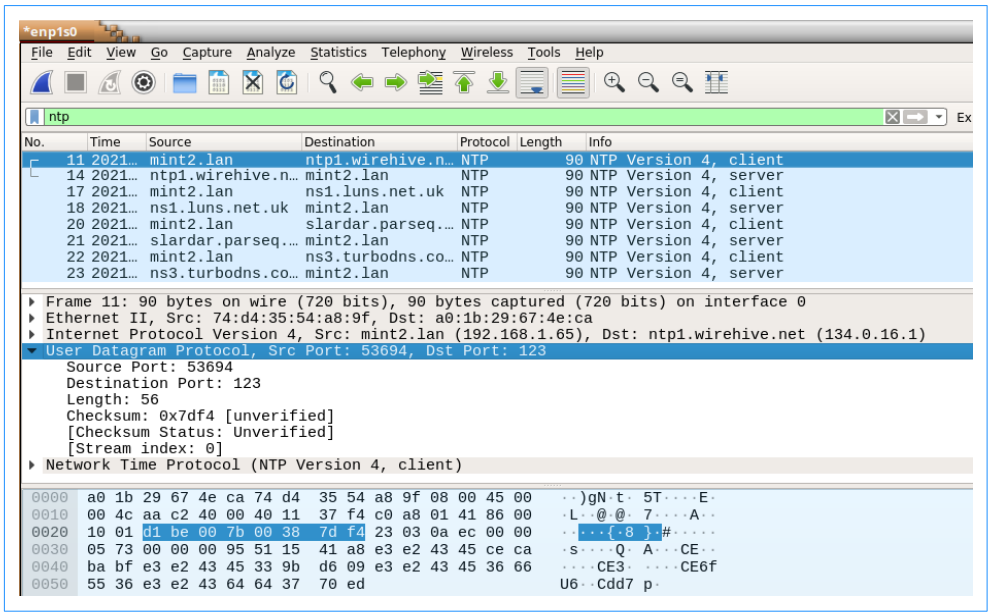
Clicking the next line, Internet Protocol Version 4, shows the bytes added by the IP protocol:



Fields in this include the source and destination address. The source, for example, is mint2.lan, with IP address 192.168.1.65, which in hex is c0 a8 01 41.

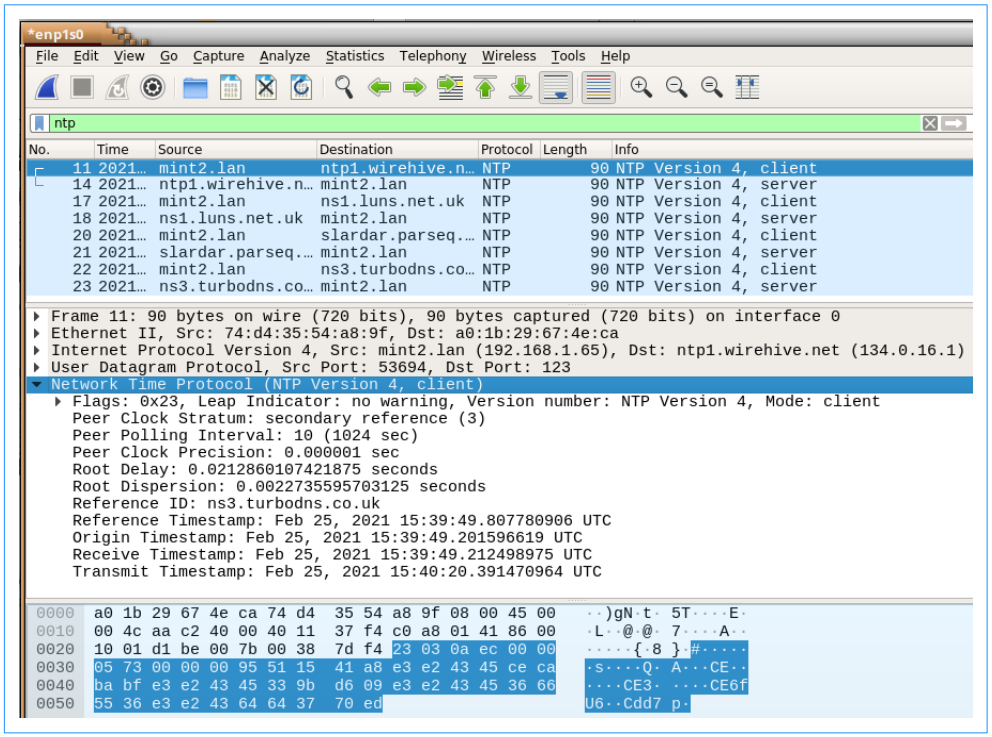
Another field is the Time to live, 64. If the packet is sent on 64 router hops it will stop - to ensure it will not bounce around the Internet forever. The protocol is UDP.

Clicking this line:



shows the UDP fields added - port numbers and others.

Finally the core NTP protocol:



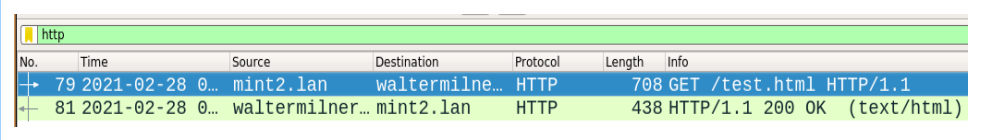
This shows the time fields, and shows this server was stratum 3.

This shows how the 90 byte frame was constructed. It starts with the fields from the NTP protocol. UDP adds port numbers. IP adds source and destination addresses. The link layer protocol adds MAC addresses for the first link to the gateway.

HTTP

This is hyper-text transfer protocol, the application layer protocol used by the 'web'.

Suppose in teh address bar of a web browser someone types waltermilner.com/test.html, while WireShark is filtering http packets. We get:



We simply get a request and a response packet. The request is a GET, asking for the file test.html, and the response is a header saying 200, OK, and a payload which is the file test.html.

If we look at the application layer fields of the first packet, we see:

Hypertext Transfer Protocol
GET /test.html HTTP/1.1\r\n
Host: waltermilner.com\r\n
Connection: keep-alive\r\n
Pragma: no-cache\r\n
Cache-Control: no-cache\r\n
Upgrade-Insecure-Requests: 1\r\n
User-Agent: Mozilla/5.0 (BB10; Touch) AppleWebKit/537.10+ (KHTML, like Gecko) Version/10.0.9.2372 Mobile Safari/537.10+\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9\r\n
Accept-Encoding: gzip, deflate\r\n
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8\r\n
Cookie: timezone=Europe/London; horde_sidebar_c_nag-toggle-shared=0; Horde=83258a2359e20ba31eca31bb9328a1cb; horde_secret_key=eG4dMh2l95r4Vugsty_UlcE\r\n
\r\n

Each line ends \r\n, which are the control characters for <return> and <newline>.

The first two lines are the request, to fetch teh file test.html, from server waltermilner.com.

Connection: keep-alive\r\n asks the server to keep open the TCP connection. These http packets are being sent over an established TCP. We do not want it closed, then a new one establishe dto send back the page.

Pragma: no-cache\r\n Cache-Control: no-cache\r\n means do not use a cache, so we can try to do this more than once and get the same result (set in the Chrome browser More tools.. Developer tools.. Disable cache)
Upgrade-Insecure-Requests: 1\r\n tells teh server it can switch to a secure site version - probably https://waltermilner.com
User-Agent: Mozilla/5.0 (BB10; Touch) AppleWebKit/537.10+ (KHTML, like Gecko) Version/10.0.9.2372 Mobile Safari/537.10+\r\n tells the server what user-agent is sending teh request. The user agent is often a web browser, but it might be a bot, possibly from a search engine.

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9\r\n. This tells the server what file types the sender can handle.

Accept-Encoding: gzip, deflate\r\n. This tells the server what compressed formats the sender can handle.

Accept-Language: en-GB,en-US;q=0.9,en;q=0.8\r\ indicates what languages the browser can 'understand' - the 'locale setting'

Cookie: timezone=Europe/London; horde_sidebar_c_nag-toggle-shared=0; Horde=83258a2359e20ba31eca31bb9328a1cb; horde_secret_key=eG4dMh2l95r4Vugsty_UlcE\r\n is the set of cookie name-value pairs.

The http packet sent back in response (frame 81 in this example) has a header with a set of fields similar to this, followed by 12 lines of text based data, which is the actual web page test.html.

GET and POST

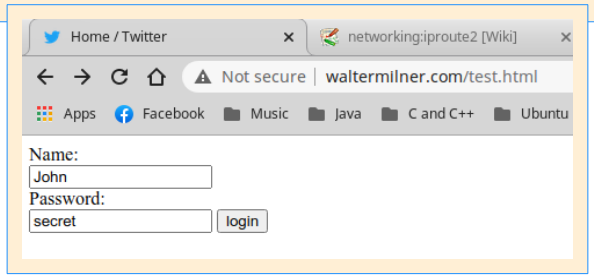
Suppose we change the test.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Computer Science - test 2</title>
</head>
```

```
<body>
  <form method="GET" action="login.php">
     <label for="name">Name:</label><br>
     <input type="text" id="name" name="name"><br>
     <label for="password">Password:</label><br>
     <input type="text" id="password" name="password">
     <input type="submit" value="login">
  </form>
</body>

</html>
```

which renders as:



When the user clicks the submit button, the browser will send off this data to the script login.php.

If we look at what http packet this actually sends, we see:

Hypertext Transfer Protocol

GET /login.php?name=John&password=secret HTTP/1.1\r\n
Host: waltermilner.com\r\n
Connection: keep-alive\r\n
Pragma: no-cache\r\n..

so method=get means the form data is sent as name-value pairs, in the 'query string' in the URL - that is, /login.php?name=John&password=secret.

That means the password s sent as plain text in the URL, and this is totally insecure.

If we change the web page to use POST method:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Computer Science - test 2</title>
</head>

<body>
<form method="POST" action="login.php">
..
</body>

</html>
```

The http packet header this sends is:

Hypertext Transfer Protocol

POST /login.php HTTP/1.1\r\n
Host: waltermilner.com\r\n
Connection: keep-alive\r\n

Content-Length: 25\r\n..

but after the header is data:

HTML Form URL Encoded: application/x-www-form-urlencoded

and the data bytes are:

6e 61 6d 65 3d 4a 6f 68 6e 26 70 61 73 73 77 6f

72 64 3d 73 65 63 72 65 74

which are the code points of "name=John&password=secret"

so POST is almost as insecure as GET.

So how do you secure private data? Suggestions:

- Salt and hash data with JavaScript before sending
- Send using POST, and
- Use https not http, so data is additionally encrypted for transmission.

But for good information security - do not buy a computer.

Socket Programming

The idea of a socket is to have a sockets on computers which you can plug cables into to connect them and exchange data. But these are software sockets, not hardware ones, and the cable is in fact the Internet. By using lower level protocol code, we can ignore the network interfaces, different media, switches and routers in between the two machines.

The idea was first used in a version of Unix from Berkeley, so thes eare now called Berkeley sockets or BSD sockets. They have now evolved into a POSIX standard. The reference documentation is here

Struct and functions

Ideally we would have objects like client and server, and methods like connect and send. But this is C and there is no OOP, so we have some structs defined as useful data structures, and functions to do this, as defined or declared in some header files.

header files

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

structs

```
struct sockaddr_in { // IP address, port and family eg IPv4
    short            sin_family;   // e.g. AF_INET, IPv4
    unsigned short   sin_port;     // port number
    struct in_addr   sin_addr;     // protocol address
    char             sin_zero[8];  // for system use
};

struct in_addr { // IP address as a long
    unsigned long s_addr;  // just a long really
};
```

functions

socket() - create a socket

bind() - associate a socket with an address

listen() - tell a bound socket to listen for connections

accept() - return a new socket, as response to a bound listening socket

recv() - receive a message from a connected socket

send() - send a message to a connected socket

shutdown() - shutdown a socket

htons() - convert a port int to network byte order

Fetching a web page

The first step is to do just one side - code to send a GET request to an http server, and receive the web page sent back.

In outline, the code:

1. Creates a socket
2. Connects to an http server
3. Sends a GET command
4. Recieves the reply
5. Outputs it

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

int main(void)
{
  int socketID;
```

```
  struct sockaddr_in server;

  printf("Started\n");

  //Create socket
  socketID= socket(AF_INET , SOCK_STREAM , 0);
  if (socketID == -1)
  {
    printf("Could not create socket");
    return 1;
  }

   //Connect to remote server
  server.sin_addr.s_addr = inet_addr("93.184.216.34");
  server.sin_family = AF_INET;
  server.sin_port = htons( 80 );

  if (connect(socketID , (struct sockaddr *)&server , sizeof(server)) < 0)
  {
    puts("connect error");
    return 1;
  }
  puts("Connected");

  // Send GET request
  // HTTP 1.1 requires a Host header
  char * message = "GET / HTTP/1.1\r\nHost: example.com\r\n\r\n";
  if( send(socketID , message , strlen(message) , 0) < 0)
  {
    puts("Send failed");
    return 1;
  }
  puts("Data Send\n");

  //Receive a reply from the server
  char reply[2000];
  if( recv(socketID, reply , 2000 , 0) < 0)
  {
    puts("recv failed");
  }
  puts("Reply received\n");
  puts(reply);
  close(socketID);

  return 0;
}
```

The socket function creates a socket. It takes 3 parameters- the protocol family, the type, and the protocol. Here we have

```
socketID= socket(AF_INET , SOCK_STREAM , 0);
```

with AF_INET meaning IPv4, SOCK_STREAM meaning TCP, and so connection-based, and protocol 0, allowing the system to choose it - so it will be TCP. it returns an int, to identify the socket.

Then connect is a function to connect a socket. It has 3 parameters - the socket, the address of the server in a struct, and the length of the struct. So

```
connect(socketID , (struct sockaddr *)&server , sizeof(server)
```

The struct has fields for the protocol family (AF_INET, IPv4), the port number (which will be 80) and the address, which is a long:

```
server.sin_addr.s_addr = inet_addr("93.184.216.34");
server.sin_family = AF_INET;
server.sin_port = htons( 80 );
```

The function inet_addr converts a string in dotted notation into a long.

We can then send the message, by

```
send(socketID , message , strlen(message) , 0)
```

The fourth parameter is a flag of OR'd bits, with meaninsg which depend on the protocol. 0 works.

Function recv corresponds to send in reverse:

```
recv(socketID, reply , 2000 , 0)
```

Here we have set up reply as a buffer of 2000 chars.

Send and receive a message

The next step is to have code which will accept and output a received message, and a sender to send it. The sender will send to 127.0.0.1 so the same machine, and the receiver needs to be started running first. The sender is:

```
 // connect and send a message
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

int main(void)
{
  int socketID;
  struct sockaddr_in server;

  printf("Sender started\n");

  //Create socket
  socketID= socket(AF_INET , SOCK_STREAM , 0); // TCP
  if (socketID == -1)
  {
    printf("Could not create socket");
    return 1;
  }

  //Prepare to connect
  server.sin_addr.s_addr = inet_addr("127.0.0.1"); // local host
  server.sin_family = AF_INET;
  server.sin_port = htons( 1234 ); // port 1234

  if (connect(socketID , (struct sockaddr *)&server , sizeof(server)) < 0)
  {
    puts("connect error");
    return 1;
  }
```

```
  puts("Connected");


  char * message = "Hello";
  if( send(socketID , message , strlen(message) , 0) < 0)
  {
    puts("Send failed");
    return 1;
  }
  puts("Data Sent\n");

  close(socketID);

  return 0;
}
```

and the receiver is:

```
// recieve a message and display it
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

int main(int argc , char *argv[])
{
  int socketIn , socketOut , c;
  struct sockaddr_in server , client;

  //Create a socket
  socketIn = socket(AF_INET , SOCK_STREAM , 0);
  if (socketIn == -1)
  {
    printf("Could not create socket");
  }

  //Prepare the sockaddr_in structure
  server.sin_family = AF_INET; // IPv4
  server.sin_addr.s_addr = INADDR_ANY; // will bin dto all interfaces
  server.sin_port = htons( 1234 ); // using port 1234

  //Bind  - associate socket with this address
  if( bind(socketIn,(struct sockaddr *)&server , sizeof(server)) < 0)
  {
    puts("bind failed");
    return 1;
  }
  puts("bind done");

  //Listen
  listen(socketIn , 1); // allow a backlog of only 1 outstanding connection

  //Accept an incoming connection
  puts("Waiting for incoming connections...");
  c = sizeof(struct sockaddr_in);
  socketOut = accept(socketIn, (struct sockaddr *)&client, (socklen_t*)&c);
  // socketOut is a socket which can receive from where socketIn is coming from
  if (socketOut<0)
  {
    perror("accept failed");
    return 1;
```

```
  }

  puts("Connection accepted");

  #define BUFFER_LENGTH 1000
  char message[BUFFER_LENGTH];

  if( recv(socketOut, message , BUFFER_LENGTH , 0) < 0)
  {
    puts("recv failed");
  }
  else
  {
    puts(message);
  }

  close(socketIn);
  close(socketOut);

  return 0;
}
```