http

Protocols

This section is about http and its background ideas.

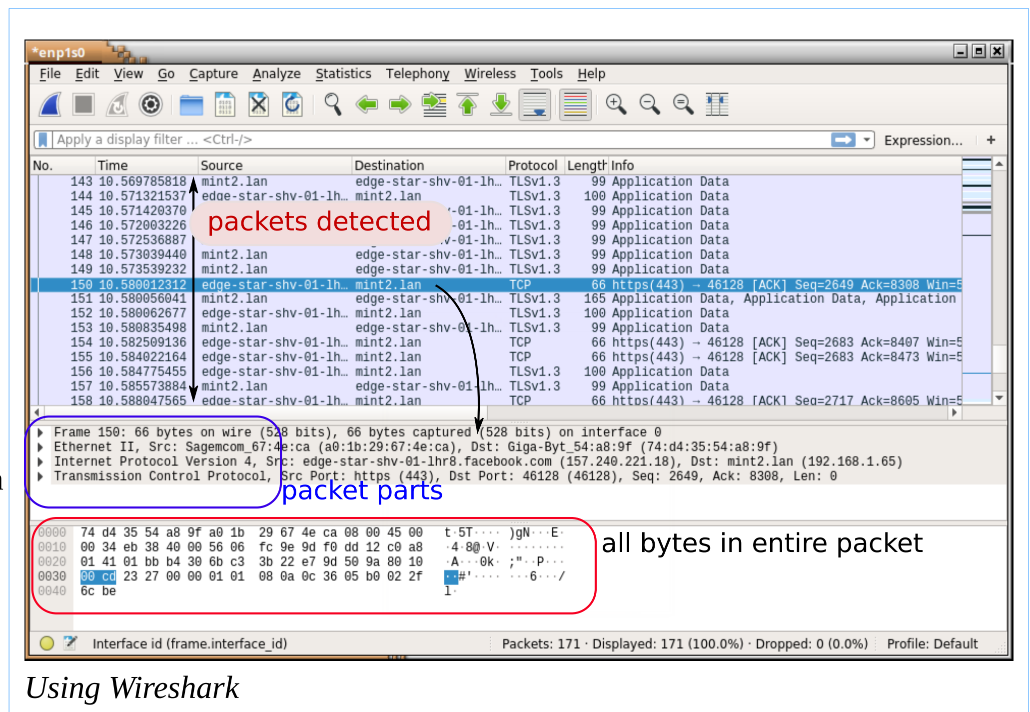
The Internet is a set of computing devices which send messages to each other. These messages are used for many purposes. The Web aka the world-wide web is one such service. Others are email, file transfer, time synchronisation and so on.

Http stands for Hyper-Text Transfer Protocol. A protocol is a set of rules or conventions which are needed for communication, so that the sender knows what to send, in what order, and the receiver knows what to expect.

On the Internet messages are sent in data bursts called packets.

Packet sniffing

A packet sniffer is a piece of software which reads and displays packets a device can detect. Wireshark (formerly known as Ethereal) is the best known, and is free open source. You may need to run it with root priveldges, so on Linux start it at the command line with sudo wireshark:



Using Wireshark

TCP/IP

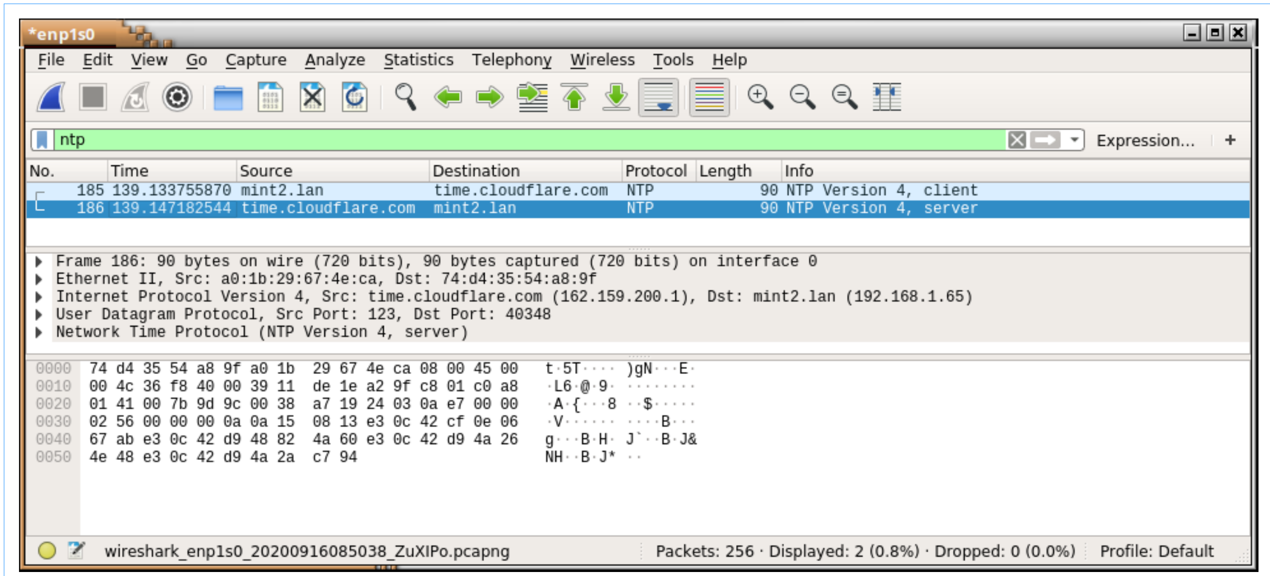
The Internet has to handle several issues:

- Connecting different devices – desktops, tablets, mobile phones, with different OS and versions on different platforms
- Through different media – cable, fibre, wireless, satellite and so on.
- Through different kinds of devices – end points, routers, gateways and so on.
- For different purposes – web pages, email, VOIP and so on.
- Resilience – it should continue to work if some links are very busy with heavy traffic, or some routers fail completely

These problems are handled by using a set of different protocols, arranged in layers. This is sometimes called a protocol stack. The Internet uses a set of protocol layers called TCP/IP.

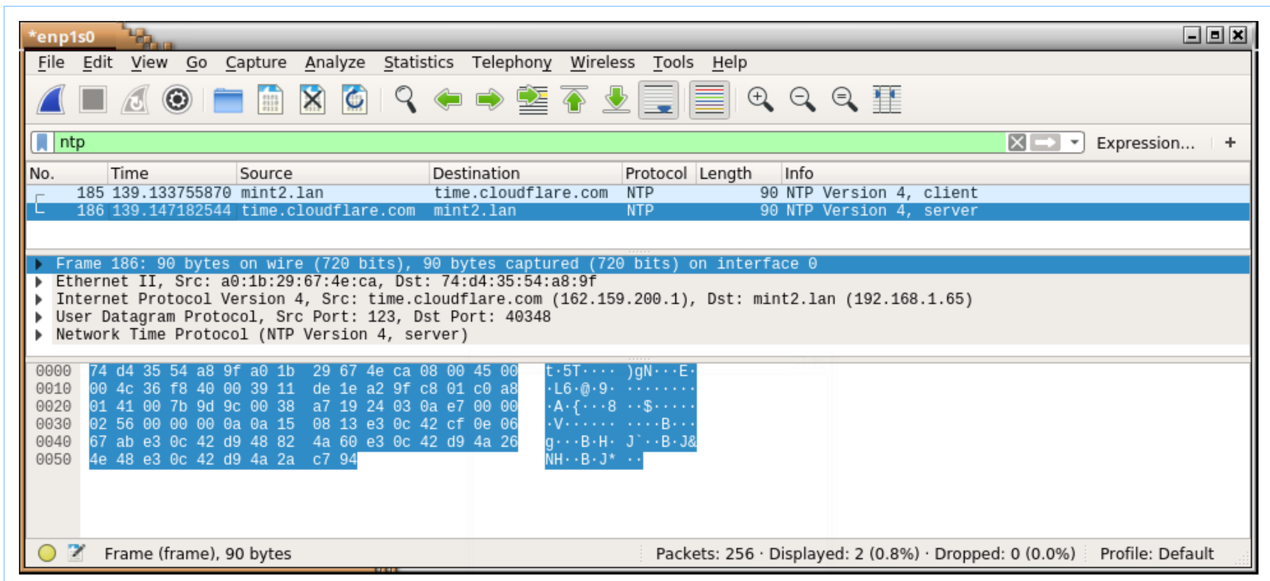
NTP

To illustrate the idea with a practical example, we use ntp. This is Network Time Protocol, a way to get accurate time settings on networked machines. This is Wireshark capturing ntp packets:



Here, mint2.lan is the desktop I am writing this on, and time.cloudflare.com is a remote time server which is offering the time check service. Two packets have been captured. One is from the client to the server, asking what the time is, and the other is server to client, supplying the time.

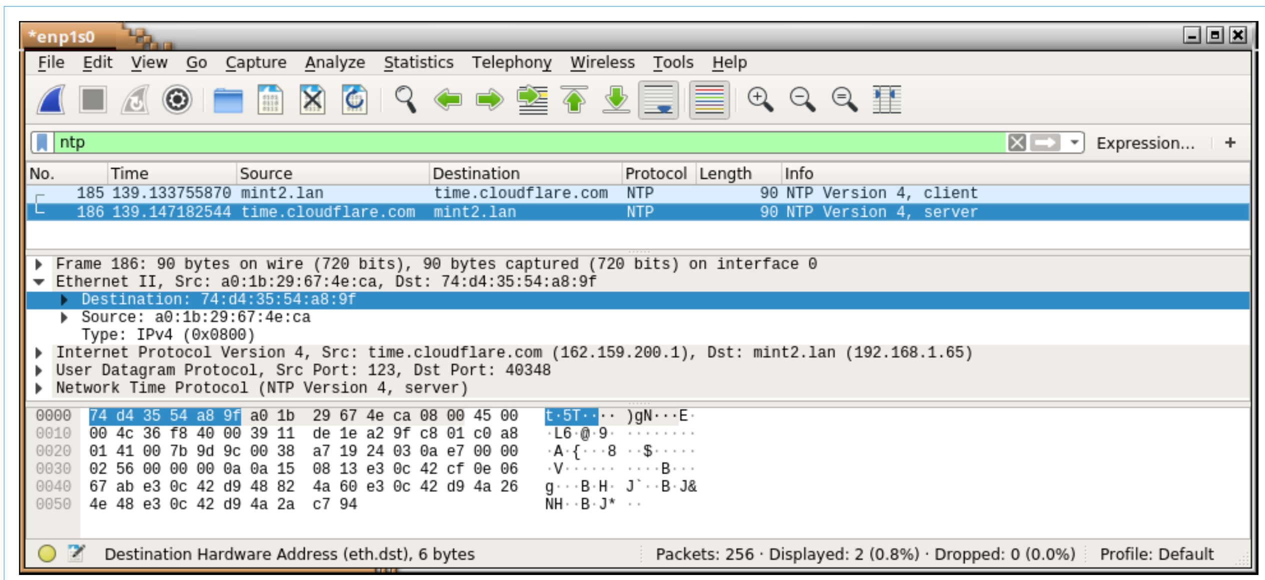
This shows the second packet, server to client:



The frame was 90 bytes long, on the ‘wire’. Interface 0 is an Ethernet cable connection, so this really is a wire, not wireless or fibre (in the local link). The lower section, starting 74 and ending 94, are the bytes, in hex, in that frame.

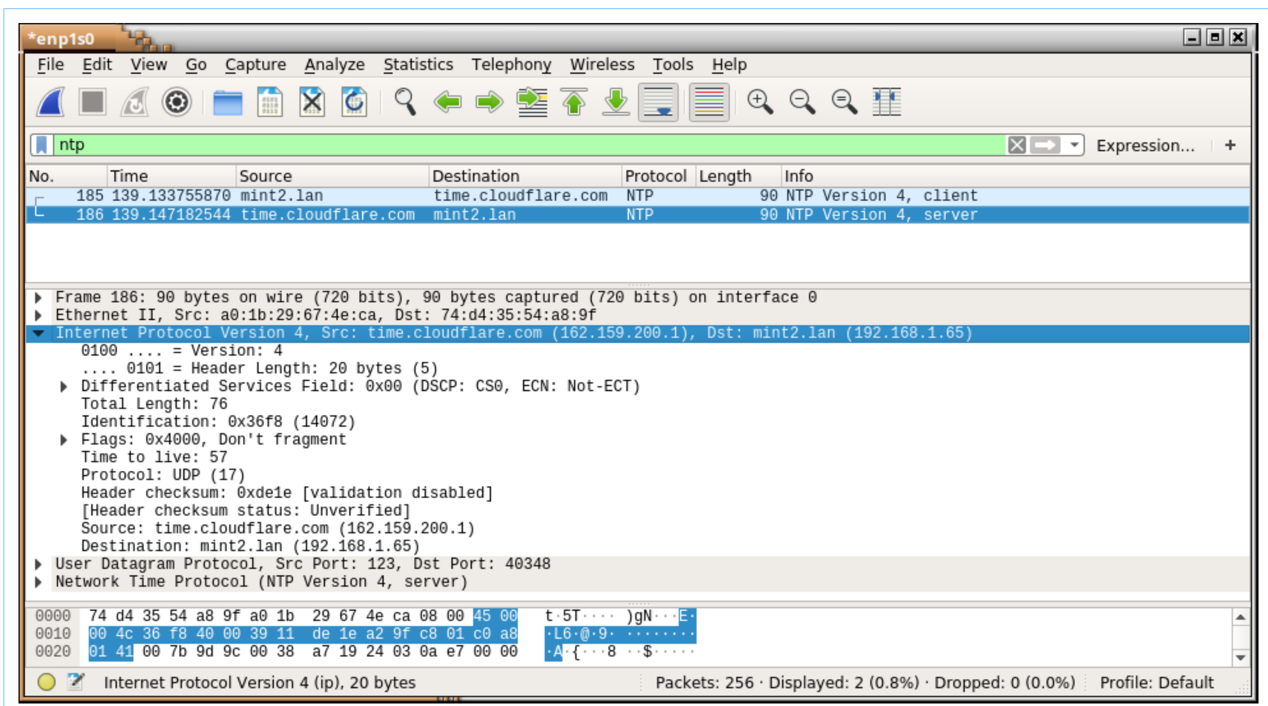
In the middle section it also lists Ethernet II, Internet Protocol, User Datagram Protocol and Network Time protocol. The 90 bytes of the frame are made up of these 4 parts, which match the 4 layers of the TCP/IP scheme.

The Ethernet II is the ‘link layer’:



This has a destination and source address – for example the destination is 74:d4:35:54:a8:9f. This is the MAC address of this computer – the address hard-wired into the network interface electronics in this machine.

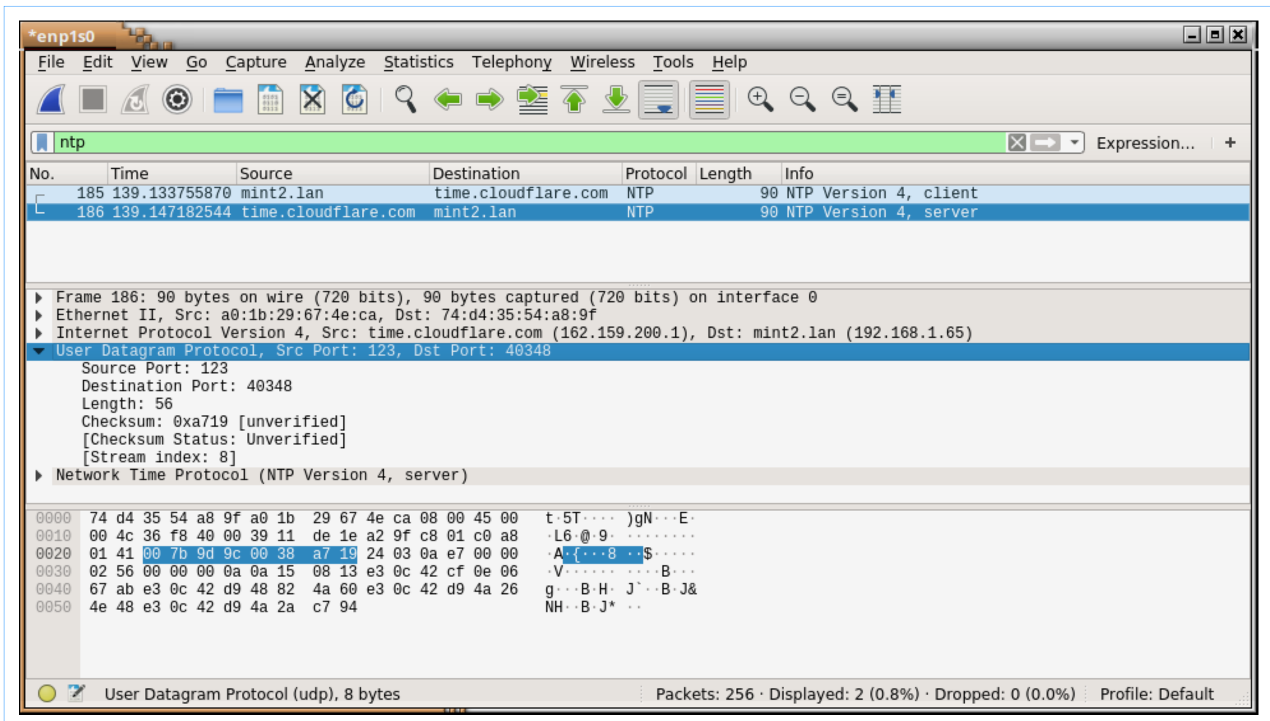
The second block of bytes is for the Internet Protocol packet, which is in the second TCP/IP layer:



This also has the source and destination addresses, as IP addresses (not MAC). So the source is 162.159.200.1, which is the IP address of time.cloudflare.com. Another field is ‘Time to live’, value 57. Suppose the packet cannot be delivered, maybe because someone switched off the destination. We need to ensure the packet does not bounce around the Internet forever, and the ‘time to live’

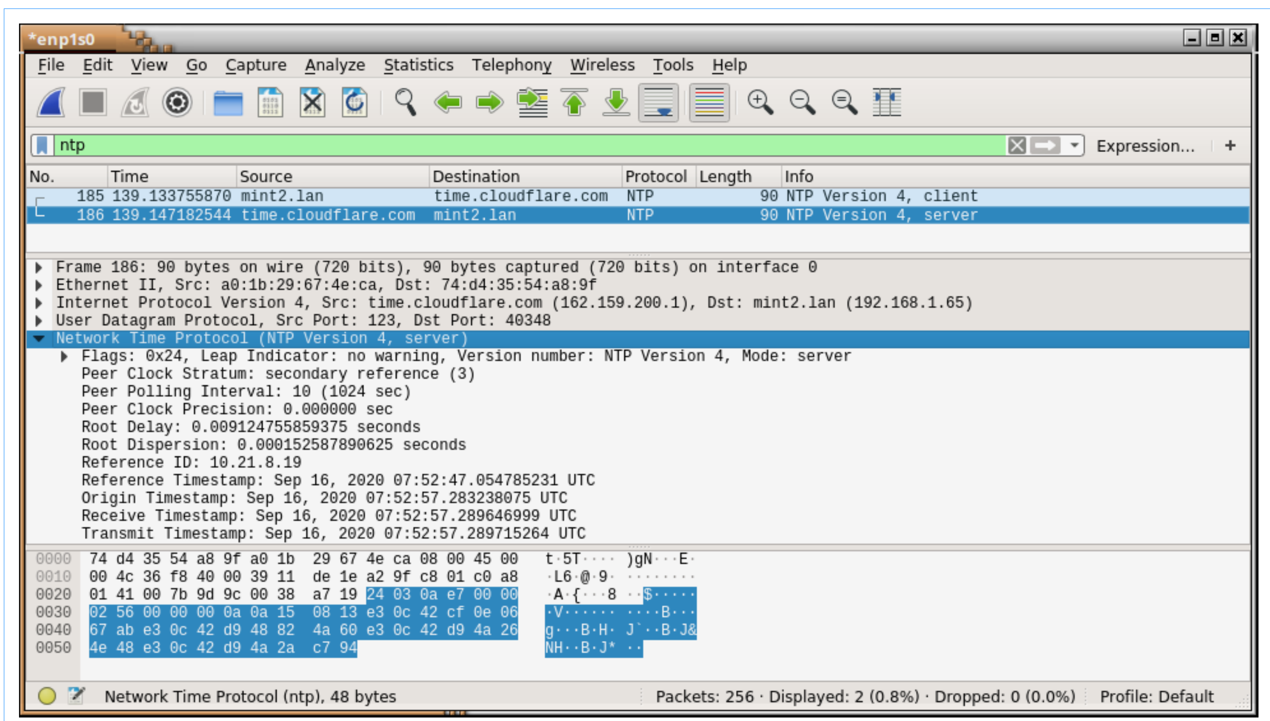
field does this. This ('the hop count') is reduced by 1 every time the packet is sent device to device, and when it counts down to 0, the packet is destroyed.

The third layer holds a User Datagram Protocol packet:



The UDP includes port numbers – through different ports one device can connect to different destinations for different purposes at the same time.

The 4th layer is the actual NTP packet:



This contains fields relevant to the NTP protocol. For example, the Peer Clock Stratum is 3. Time servers are arranged in a hierarchy. Stratum 0 are very accurate atomic clocks, not directly accessible. They are connected to stratum 1 machines, which also connect to other stratum 1

machines to check time. Stratum 2 machines connect to these, and each other, and 3 to 2. This gives a large number of time servers available, all synchronised to a small number of very accurate clocks. This is not simple, due to time delays in sending and receiving packets. The target is an accuracy of 20 pico-seconds, which is 10^{-12} seconds – around 1000 times faster than any current device.

The TCP/IP layers

TCP/IP is a set of many protocols, arranged in four layers:

Link layer – handles different media like cable, wireless and fibre, and protocols like PPP, point-to-point protocol, which many ISPs use to connect routers in home across phone lines to backbone routers. Links do not cross routers.

Internet layer – like Internet protocol versions 4 and 6, connecting routers

Transport layer- like UDP and TCP, connect client to server

Application layer – like NTP and htp and ftp and telnet

So a frame on a medium (like an Ethernet cable) would have link layer bytes, wrapping for example an IP packet, wrapping a UDP packet, wrapping an NTP packet, as in the example above.

Communication can be connectionless, like UDP. Here the client sends out a packet, with a source and destination address. The packet might be delivered – or it might not. NTP uses UDP and is connectionless.

Or the communication might be connection-based, like TCP. This involves a sequence of messages back and forth, establishing a connection, exchanging messages and closing the connection. Http in the application layer uses TCP in the transport layer and is connection-based.

For application programming, we only need to be concerned with application layer protocols, and can think of the rest as a black box through which client and server are connected.

TCP

UDP is connectionless, but TCP is connection-based – and http uses it, so we look at TCP. The idea is a 3 stage sequence:

1. Establish a connection
2. Use the connection
3. Close the connection

At each stage we need to be sure messages have been set.

For example to establish a connection, 3 packets are sent:

A SYN from the client requesting connection

A SYN/ACK from the server acknowledging the request, and sending an accept back

An ACK from the client acknowledging the accept.

So for example a packet from the server back to the client is:

```
Frame 61: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
Ethernet II, Src: a0:1b:29:67:4e:ca, Dst: 74:d4:35:54:a8:9f
Internet Protocol Version 4, Src: waltermilner.com (87.117.220.232), Dst: mint2.lan (192.168.1.65)
Transmission Control Protocol, Src Port: 80, Dst Port: 56300, Seq: 0, Ack: 1, Len: 0
    Source Port: 80
    Destination Port: 56300
    [Stream index: 3]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    [Next sequence number: 0    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    1010 .... = Header Length: 40 bytes (10)
    Flags: 0x012 (SYN, ACK)
        000. .... .... = Reserved: Not set
        ...0 .... .... = Nonce: Not set
        .... 0... .... = Congestion Window Reduced (CWR): Not set
        .... .0.. .... = ECN-Echo: Not set
        .... ..0. .... = Urgent: Not set
        .... ...1 .... = Acknowledgment: Set
        .... .... 0... = Push: Not set
        .... .... .0.. = Reset: Not set
        .... .... ..1. = Syn: Set
        .... .... ...0 = Fin: Not set
        [TCP Flags: ·······A··S·]
    Window size value: 28960
    [Calculated window size: 28960]
    Checksum: 0xf958 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP),
Window scale
    [SEQ/ACK analysis]
        [This is an ACK to the segment in frame: 57]
        [The RTT to ACK the segment was: 0.013444470 seconds]
        [iRTT: 0.013524670 seconds]
    [Timestamps]
```

where frame 57 was requesting the connection

Http

Http is one application layer protocol that operates over TCP.

For example a request for a web page sent by a browser to a server would be this TCP packet

```
Frame 63: 555 bytes on wire (4440 bits), 555 bytes captured (4440 bits) on interface 0
Ethernet II, Src: 74:d4:35:54:a8:9f, Dst: a0:1b:29:67:4e:ca
Internet Protocol Version 4, Src: mint2.lan (192.168.1.65), Dst: waltermilner.com (87.117.220.232)
Transmission Control Protocol, Src Port: 56300, Dst Port: 80, Seq: 1, Ack: 1, Len: 489
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 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/85.0.4183.102 Safari/537.36\r\n
    Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,ap
plication/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
    \r\n
    [Full request URI: http://waltermilner.com/test.html]
    [HTTP request 1/2]
    [Response in frame: 67]
    [Next request in frame: 75]
```

This is requesting test.html from the server at waltermilner.com. The \r\n are control characters for a RETURN and LINE FEED. The http command is GET.

The server sends back:

```
 Frame 67: 424 bytes on wire (3392 bits), 424 bytes captured (3392 bits) on interface 0
Ethernet II, Src: a0:1b:29:67:4e:ca, Dst: 74:d4:35:54:a8:9f
Internet Protocol Version 4, Src: waltermilner.com (87.117.220.232), Dst: mint2.lan (192.168.1.65)
Transmission Control Protocol, Src Port: 80, Dst Port: 56300, Seq: 1, Ack: 490, Len: 358
Hypertext Transfer Protocol
    HTTP/1.1 200 OK\r\n
    Connection: Keep-Alive\r\n
    Content-Type: text/html\r\n
    Last-Modified: Fri, 11 Apr 2014 13:21:50 GMT\r\n
    Accept-Ranges: bytes\r\n
    Content-Length: 145\r\n
    Date: Wed, 16 Sep 2020 14:42:48 GMT\r\n
    Server: LiteSpeed\r\n
    \r\n
    [HTTP response 1/2]
    [Time since request: 0.015230382 seconds]
    [Request in frame: 63]
    [Next request in frame: 75]
    [Next response in frame: 76]
    [Request URI: http://waltermilner.com/test.html]
    File Data: 145 bytes
Line-based text data: text/html (9 lines)
    <!DOCTYPE html>\n
    <html>\n
        <head>\n
            <title>Title</title>\n
        </head>\n
        <body>\n
            <div>This is a test page</div>\n
        </body>\n
    </html>\n
```

The http packet contains a header, in yellow, ended by a \r\n blank line, then the actual web page, in blue. This is http response 1 of 2, because the browser also asks for the favicon:

```
75                     10.264678467  mint2.lan      waltermilner.com      HTTP    486     GET
/favicon.ico HTTP/1.1
```

which the server sends back as an image:

```
76                     10.277974882  waltermilner.com      mint2.lan      HTTP    1768    HTTP/1.1
200 OK  (image/x-icon)
```

http headers

The above examples show how an http request or response packet can contain header fields with values, and these provide the server or client with additional information. Details are documented here. Some notes:

Connection: keep-alive\r\n. The 'connection' means the TCP connection. Keep-alive means the browser expects it might request other pages, and does not want to go through another connection setup sequence.

Pragma: no-cache\r\n

Cache-Control: no-cache\r\n. A cache means a copy of the requested item. So a server might keep a copy of a requested web page, maybe in memory, to make it faster to find it again for another request. Or a browser might save a copy of a web page locally, client-side, so it does not have to fetch it again on another view request. Pragma is a deprecated version

User-Agent: .. The user agent is the software which is making the request. Usually it is some browser, but it might be a web-searching robot – or anything else. This header supplies information on that.

Accept: text/html,application/xhtml..;q=0.9,image/avif,.. says what type of data is expected to be returned – html as text, or a graphics image in binary format a dn so on. The q=.. provides an indication of which is preferred.

Last-Modified: Fri, 11 Apr 2014 13:21:50 GMT\r\n – tells the browser when the file on the server was last modified

Date: Wed, 16 Sep 2020 14:42:48 GMT\r\n – tells the browser the date and time on the server

Server: LiteSpeed – tells the browser what server software is being used.

Http requests and responses

An http request packet will usually be sent from browser to server. It will consist of a request, some headers as above, and a data body, depending on what the request is.

The common requests are GET and POST. These are sometimes confused. The intention was that GET is used to *fetch* a resource – usually a web page maybe with other things like images, from server to browser, while POST would be used to *send* something to the server, to do something such as add a record to a database or login. The POST would often send name-value pairs inputted through an html form, such as

```
 <!DOCTYPE html>

<html>
    <head>
        <title>The title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <form method="POST" action="login.php">
            <input type="text" id="fname" name="fname"><br>
            <input type="text" id="lname" name="lname">
            <input type="submit" value="Send">
        </form>
    </body>
</html>
```

When John Brown is entered for the name and the Send button is clicked, an http packet is sent :

```
Frame 1: 796 bytes on wire (6368 bits), 796 bytes captured (6368 bits) on interface 0
Ethernet II, Src: 00:00:00:00:00:00, Dst: 00:00:00:00:00:00
Internet Protocol Version 4, Src: localhost (127.0.0.1), Dst: localhost (127.0.0.1)
Transmission Control Protocol, Src Port: 46110, Dst Port: 80, Seq: 1, Ack: 1, Len: 730
Hypertext Transfer Protocol
    POST /login.php HTTP/1.1\r\n
    Host: 127.0.0.1\r\n
    Connection: keep-alive\r\n
    .. other headers..
    Accept-Language: en-GB,en-US;q=0.9,en;q=0.8\r\n
    \r\n

    File Data: 22 bytes
HTML Form URL Encoded: application/x-www-form-urlencoded
    Form item: "fname" = "John"
    Form item: "lname" = "Brown"
```

So this starts POST /login.php, followed by headers, then we have a data block, consisting of names (like fname) and values (like John).

On the server, login.php would be a script which would actually do the login, and output html, which would be sent back to the browser. Login.php would be a PHP script, which is easily able to pick up the name-value pairs and process them as required, with simple database access.

The confusion comes about because GET will be a pretty similar thing. If we change the method to GET, this packet is sent:

```
Frame 7: 677 bytes on wire (5416 bits), 677 bytes captured (5416 bits) on interface 0
Ethernet II, Src: 00:00:00:00:00:00, Dst: 00:00:00:00:00:00
Internet Protocol Version 4, Src: localhost (127.0.0.1), Dst: localhost (127.0.0.1)
Transmission Control Protocol, Src Port: 46146, Dst Port: 80, Seq: 1, Ack: 1, Len: 611
Hypertext Transfer Protocol
    GET /login.php?fname=another&lname=name HTTP/1.1\r\n
    Host: 127.0.0.1\r\n
    .. other headers
    Accept-Language: en-GB,en-US;q=0.9,en;q=0.8\r\n
    \r\n
```

This also sends the name-value pairs, but in a different way – in the ‘query string’. The URL requested is now login.php?fname=another&lname=name, as a ? followed by name=value pairs, separated by a &.

So GET can also be used to send data – but that was not the intention. GET was intended to fetch data, and POST to send.

It is sometimes said GET is insecure but POST is secure. Clearly this is not true. POST data is sent in the packet body, and can be read by anyone with a packet sniffer. So *POST is not secure*. We need to use SSL and https to send confidential data over the web.