SYS2 (NET) Laboratory 2 : HTTP, SMTP and Wireshark

These weekly lab scripts have been written as a self-study resource i.e. an activity to be worked on at your own pace, in your own time. The timetabled practical sessions are an opportunity to get advice and guidance. Therefore, you may not always finish each lab during the timetabled session, but do finish each lab's tasks in your own time. Hardware labs are open Mon-Fri, 9:00-17:00.

The aim of this lab is to explore some of the common Internet protocols used in the application layer, shown in figure 1. Before starting this lab make sure you have watched video **Lecture 2A**. Each protocol discussed in this lecture has been designed for a specific task, but are typically based around the client / server model. In this configuration you as the user (client) request a service from a remote host (server) that is either connected to your local network (LAN) e.g. a printer, or the Internet e.g. a web server. Therefore, to allow the client and server to interact we need to define a protocol, a language, to allow the two computers to request and exchange information specific to the task being performed. Typically this back and forth communications between hosts goes unseen, hidden by the application's Graphical User Interface (GUI). However, surprisingly its quite common that this communications is performed using plain text messages i.e. using human readable text. Therefore, to get a better understand of how these protocols work we will request web services using the command line, rather than your favourite web browser :). At the end of this practical you will understand how:

- The client / server model of communication works and how their protocols operate.
- HyperText Transfer Protocol (HTTP) operates and how HTML objects are transferred.
- Simple Mail Transfer Protocol (SMTP) operate and how emails are transferred.
- Wireshark can be used to examine the flow of network packets between hosts.

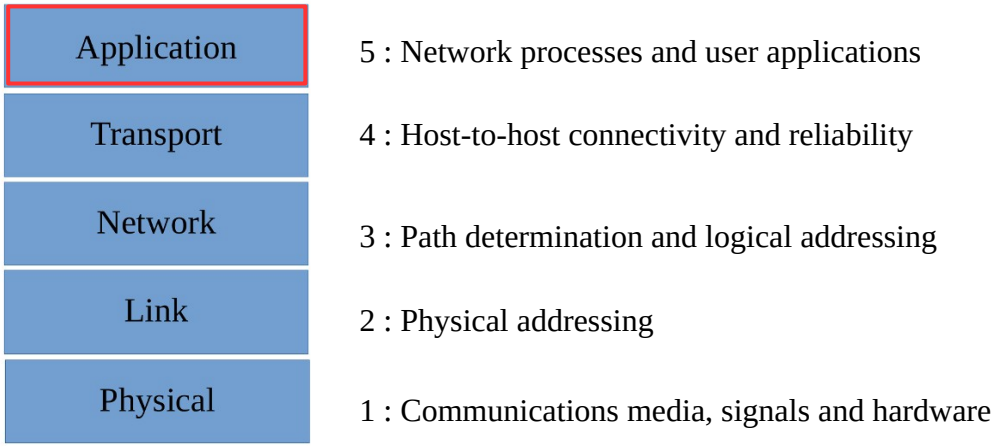


Figure 1 : The Internet protocol stack

When I was young the "Internet" was for the select few, inhabited by newsgroups and bulletin boards. For those who do not know what these are :)

https://en.wikipedia.org/wiki/Usenet_newsgroup
https://en.wikipedia.org/wiki/Bulletin_board_system

These digital resources were the discussion forums of their time, but it wasn't really until the 1990's

that things started to take off and the Internet started to become the thing we know today i.e. the world wide web (WWW). Powering this revolution were application layer protocols such as:

- HTTP : Hypertext Transfer Protocol
- SMTP : Simple Mail Transfer Protocol
- POP : Post Office Protocol
- FTP : File Transfer Protocol (we will look at this one in another lab)

Each protocol was designed to fulfil a different task and therefore take different approaches to solving the problem of how the client and server should communicate. However, at their heart they all define a protocol: a set of commands and responses which we shall now investigate.

Task 1

One of the most commonly used protocols on the Internet is the HyperText Transfer Protocol (HTTP), not to be confused with the HyperText Mark-up Language (HTML).

Note, remember HTML defines what you see on a web browser e.g. <h1> headings, <p> paragraphs, <ul> lists, <img> images etc. HTTP defines how this content is transferred.

The HyperText Transfer Protocol (RFC 2616: https://tools.ietf.org/html/rfc2616), was developed by a team at CERN, lead by Tim Berners-Lee (HTTP, HTML) in the 1990s as a method of transferring documents across the Internet i.e. web pages constructed from HTML objects, JPEG, GIF images etc. The HTTP standard has gone through a series of updates over the years, reflecting the changes in size of the Internet and how websites operate:

- 1997 : HTTPv1.1 - https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
- 2015 : HTTPv2.0 - https://en.wikipedia.org/wiki/HTTP/2
- 2020 : HTTPv3.0 - https://en.wikipedia.org/wiki/HTTP/3

The HTTP protocol is based on a request / response message pair i.e. clients request an HTML object and a server sends this data or an error code, as shown in figure 2. Originally this was performed over a non-persistent connection i.e. a connectionless protocol, a connection that is closed after each object was transferred. However, to reduce latency i.e. delays in setting up network connections, later versions of HTTP use persistent connections, allowing multiple objects sent over the same link. HTTP is a stateless protocol i.e. no information is recorded on the server about previous client connections, greatly reducing the servers processing load and memory requirements.

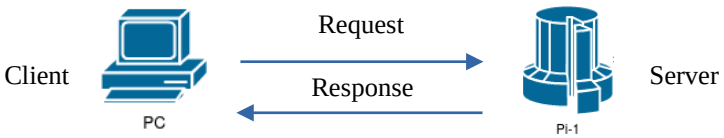


Figure 2 : Client Server model

Note, don't confuse these properties of HTTP with lower layer connection based protocols used by HTTP e.g. TCP. This transport layer protocol does have a connection state, HTTP does not. To overcome this limitation HTTP does use cookies on the client's machines to maintain session information across different pages. However, these are outside the scope of this module.

HTTP request messages (commands):

- **GET** : request a resource e.g. data, text, image etc.
- **HEAD** : same as GET, but server does not return request resource. This may sound pointless, but it is useful if you need to retrieve meta-data about a resource i.e. from the response header without having to retrieve the actual object.
- **POST** : submit data to the server e.g. send text to a specified resource etc.
- **PUT** : replace the specified resource with request message payload
- **DELETE** : as named, delete a specified resource, not commonly supported :).
- **TRACE** : loop back a message for testing.
- **OPTIONS** : returns a list of the request methods supported on this server
- **CONNECT** : for use with a network proxy / tunnel.
- **PATCH** : apply a partial modification to a specified resource

Note, these methods highlight the original aim of HTTP i.e. to transfer / share documents across an internal network. Therefore, some of these methods are quite specialised, or are now a bit of a security risk e.g. the ability to delete or modify web pages. As a general rule HTTP servers are required to implement the GET and HEAD methods, all other methods are considered optional.

Following this request message (command) there are typically a series of header fields giving details regarding the requested resource e.g. language, HTTP version, date etc. Finally there is an empty line followed by an optional message body e.g. text data for a POST request etc.

HTTP response messages return a status code and text message e.g. if a valid request was received : HTTP/1.1 200 OK, other codes fall under these general categories:

- Informational : 1XX
- Successful : 2XX
- Redirection : 3XX
- Client Error : 4XX
- Server Error : 5XX

Again, following this response message there are a series of header fields giving details regarding the requested resource e.g. media type, size, server name etc. Finally there is an empty line followed by an optional message body e.g. HTML text or image data etc.

Note, after the request and response headers you must send an empty line i.e. <CR><LF>. The <CR> character is the return ↵ key code, ASCII code 0x0D. The <LF> character is line-feed, ASCII code 0x0A. For more information of HTTP header fields and status codes refer to :

https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Every time you access a web page you tell the browser the protocol you want to use e.g. http, smtp, ftp etc. This is defined at the start of the Uniform Resource Locator (URL), as shown in figure 3. Typically the domain name maps to a specific server i.e. the server's IP address. We shall be looking at Domain Name Servers (DNS) in a later lecture / practical, for the moment we shall just ignore how a host converts the domain name into the IP address of the web server :).

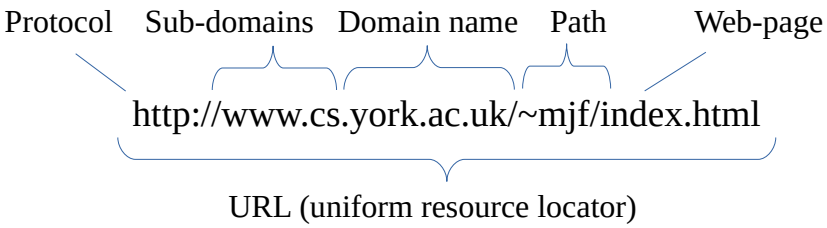


Figure 3 : an URL

A server can host multiple Internet applications, therefore, to identify what network packets should be sent to what applications (processes) running on a server, network connections are made to specific ports e.g. for HTTP we would use port 80.

IMPORTANT, a port is a software concept implemented by the host operating system, defining network end point IDs on the client and server. Ports with numbers from 0 – 1023 are system ports assigned to specific applications i.e. fixed / standardised. Port numbers ranging from 1024 – 49151 are either registered i.e. less commonly used applications, or user connections. Finally ports 49152 – 65535 are ephemeral ports i.e. dynamic ports, used to uniquely identify the client side process connecting to the server. This range does vary with OS. For more information on port numbers refer to :

https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers
https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

To view HTTP in action each Raspberry Pi is running its own web server. On the PC start your preferred browser and enter the IP address of your Pi-1 i.e. the URL below.

```
http://192.168.X.1              (X=Box/Desk)
```

Note, as always the IP address of Pi-1 is desk dependant e.g. for me X=100, refer back to lab-1 if you have forgotten how to setup / connect to your system.

If all is connected correctly you will see Bob's web page, as shown in figure 4. Enter you name and click the Submit button and you shall receive your fortune from the all knowing cow.

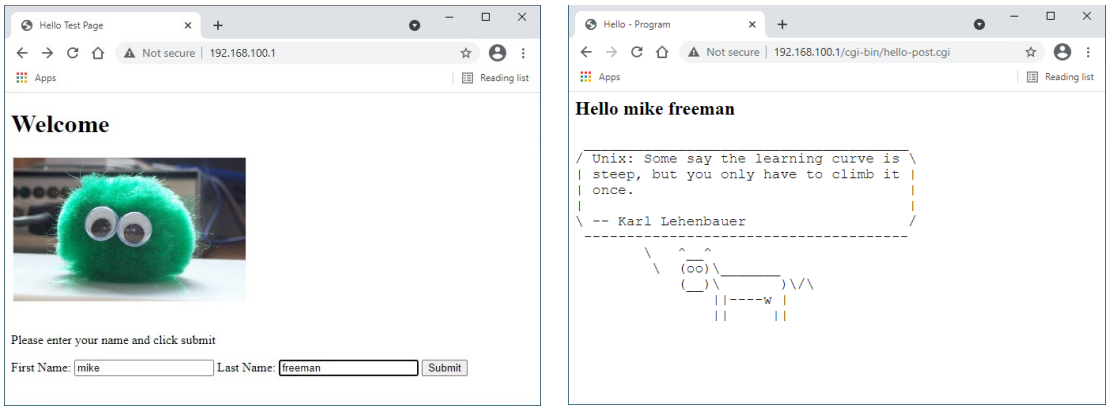


Figure 4 : Bob's webpage

Note, `fortune` and `cowsay` are both classic Linux commands, however, the fortunes told are perhaps a little dated :). Final, yes I know the CGI script is horrible, but remember Mike's programming rule number 1 : life is tooo short, things that can be bodged should be bodged :). The aim of this script was to generate variable sized / reasonable big network packets to examine later in Wireshark.

To see how these various HTML objects e.g. images and text, are transferred between the client and server we can use Telnet to manually send the required HTTP messages. To start `SSH` into Pi-2 and then from Pi-2 we will access Pi-1's web server via Telnet, as shown in figure 5.

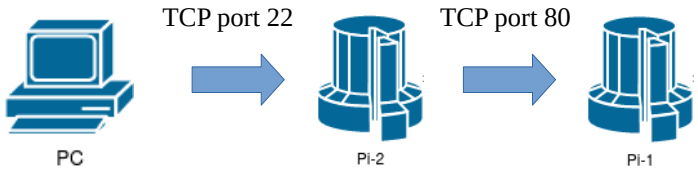


Figure 5 : connecting to Pi-1's web server

Note, normally when we use telnet we would be connecting to the Telnet server on port 23, however, in addition to specifying the IP address you wish to connect to you can also state the port.

To connect to Pi-2 open a command prompt on the PC:

`-> Command Prompt`

Within this new window start an `ssh` session with Pi-2 by entering the following command:

```
ssh pi@192.168.X.2              (X=Box/Desk)
```

This will open a terminal window on Pi-2 using default port 22. Then from Pi-2 start a Telnet session with Pi-1's web server on port 80 by entering the following command :

```
telnet 192.168.X.1 80           (X=Box/Desk, port=80)
```

This will clear the window, allowing you to enter HTTP commands. To request the root web page `/` i.e. `index.html` we can use the HTTP GET message, at the command prompt enter each line below followed by the return ↵ key:

```
GET / HTTP/1.1
Host: pi-1
```

Finally to signal the end of the GET message press the return ↵ key again i.e. enter a blank line `<CR><LF>`. The web server should then return the text stored in the file `index.html`, as shown in figure 6.

Note, the Telnet application sends keyboard character data to the specified IP address and Port. Therefore, you can **NOT** press, delete or cursor keys when typing this GET message as these characters are not defined in the HTTP protocol, and will result in a 400 Bad Request response error message.

If you enter the GET message correctly you will see the server's response message, returning the status code 200, signalling all is good, followed by a series of header fields indicating when this HTML object was last modified, its size and type etc.

Note, the request GET message may also include the types of header fields specified in the above response packet e.g. date, encoding, language (en-us), information about the client's browser and its OS, the last time the browser accessed this web page etc.

Task : using the Telnet command send a OPTIONS message to Pi-1's web server. Note, this command uses the same format as the GET message. What output does this command produce i.e. what information about the web server is displayed on the screen?

```
pi@pi-2:~ $ telnet 192.168.85.1 80
Trying 192.168.85.1...
Connected to 192.168.85.1.
Escape character is '^]'.
GET / HTTP/1.1
Host: pi-1

HTTP/1.1 200 OK
Date: Thu, 23 May 2024 14:26:58 GMT
Server: Apache/2.4.38 (Raspbian)
Last-Modified: Wed, 03 May 2023 20:16:57 GMT
ETag: "19b-5facfbcd88e85"
Accept-Ranges: bytes
Content-Length: 411
Vary: Accept-Encoding
Content-Type: text/html

<!DOCTYPE html>
<html>
<head>
<title>Hello Test Page</title>
</head>
<body>

<h1>Welcome</h1>

<img src="bob.jpg" >
<br><br>

<p>Please enter your name and click submit</p>

<form action = "/cgi-bin/hello-post.cgi" method = "post">
First Name: <input type = "text" name = "first_name" />
Last Name: <input type = "text" name = "last_name" />
<input type = "submit" value = "Submit" />
</form>

</body>
</html>
Connection closed by foreign host.
pi@pi-2:~ $
```

Figure 6 : HTTP GET request for desk 85

```
import http.client

HOST = "192.168.X.1"      #(X=Box/Desk)

conn = http.client.HTTPConnection(HOST, 80)
conn.request("GET", "/")
response = conn.getresponse()

print(response.msg)       # or response.getheaders()

print("Status:", response.status, response.reason)
print("Body:", response.read().decode())
conn.close()
```

Figure 7 : httpGet.py using `http.client` lib

```
import socket
host = "192.168.X.1"   #(X=Box/Desk)
port = 80

# Connect to the web server
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((host, port))

# Send command
request = "GET / HTTP/1.1\r\nHost: pi-1\r\n\r\n"
client.send(request.encode())

# Receive response
response = b""
while True:
    data = client.recv(1024)
    if not data:
        break
    response += data

print(response.decode())
client.close()
```

Figure 8 : httpGet.py using `socket` lib

Task 2

Using Telnet to get information about various HTML object can be a bit of a pain, therefore, to simplify this process we can write a simple python script to do this, as shown in figures 7 and 8.

Task : create a new directory "`python`" in your home folder e.g. `C:\Users\<UserName>`. Write a python program to perform the GET command previously performed using telnet. Save this code to the file: `httpGet.py` in this new directory.

Launch a python command prompt, click on the start button and select :

 `-> Anaconda`

IMPORTANT, do not use the power shell Anaconda prompt or the standard Windows command prompt CMD, as the program will not work in these.

Within the new command prompt change to the current project directory using the `cd` command. Alternatively, type "`cd `" at the command line, then within a file browser drag the current project directory into the command prompt window. Then run this program by typing:

```
python httpGet.py
```

If all is connected correctly you should again receive the same text as figure 6, as shown in figure 9.

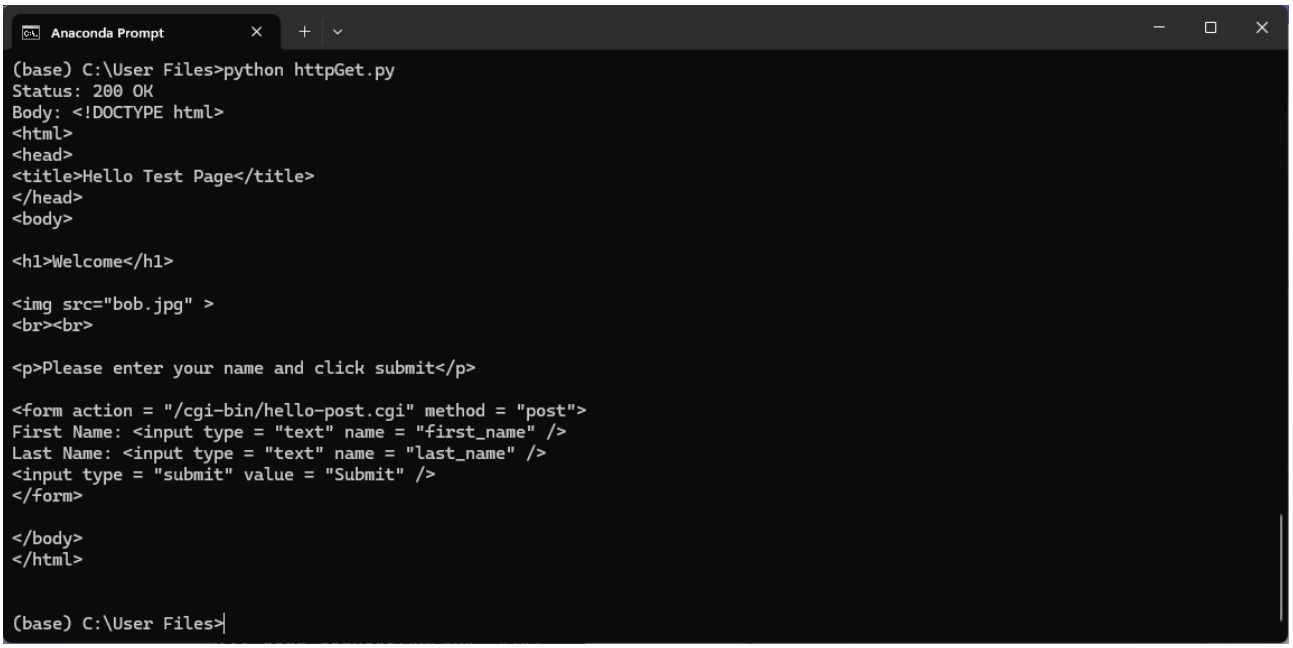


Figure 9 : HTTP GET request

Task : using the `http.client` library write a python program to send the OPTIONS message to Pi-1's web server.

Task 3

In addition to seeing the request-response text messages that are transferred between client and server we can also push deeper into the network stack and view the actual network packets that are being transferred between the Pi and PC. To capture and view network traffic associated with the HTTP request-response packets we shall be using Wireshark, more information on this software is available here:

https://www.wireshark.org/docs/

Wireshark is a network "sniffer", a piece of software that listens in on conversations between a client and its server. This software is made up of two key components, a packet capture library that operates at layer 2 in our Internet protocol stack i.e. the Link Layer, capturing all packets that are received/sent by the machine running this software. The second is a packet analyser that filters and displays the captured network packets.

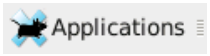
Note, a key point here is that you can only capture packets received/sent by your machine, you can not see all packets on the network.

Open a remote desktop to Pi-1 by clicking on the start button and select : `VNC Viewer`.

`-> VNC Viewer`

Next, enter the IP address of the Raspberry Pi i.e. Pi-1, that you wish to connect to and click OK. For more information on this process refer to lab 1.

To start Wireshark on the Raspberry Pi, click on the menu icon and select :



`-> Internet -> Wireshark`

This will open the Wireshark GUI as shown in figure 10. To select the network interface (NIC) to capture packets on, left click on (highlight) the Ethernet 1 (`eth1`) icon in the Interface list.

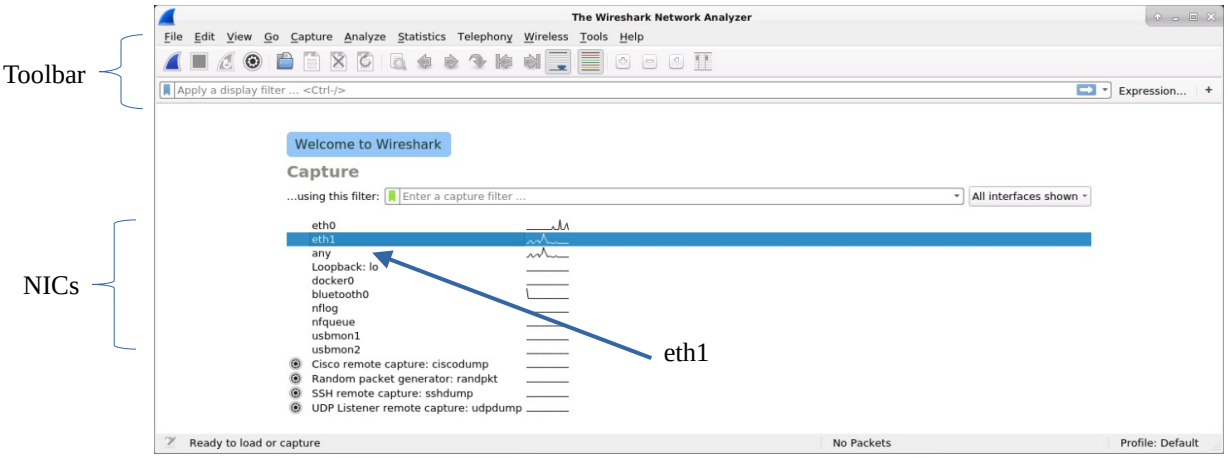


Figure 10 : Wireshark

Each Raspberry Pi has three physical network interfaces, two on-board NICs : Ethernet `eth0` and wireless interface `wlan0`. They also have an USB-to-Etherent adapter, Ethernet `eth1`, as shown in figure 11. Hardware may vary depending on when your test rig was constructed etc.

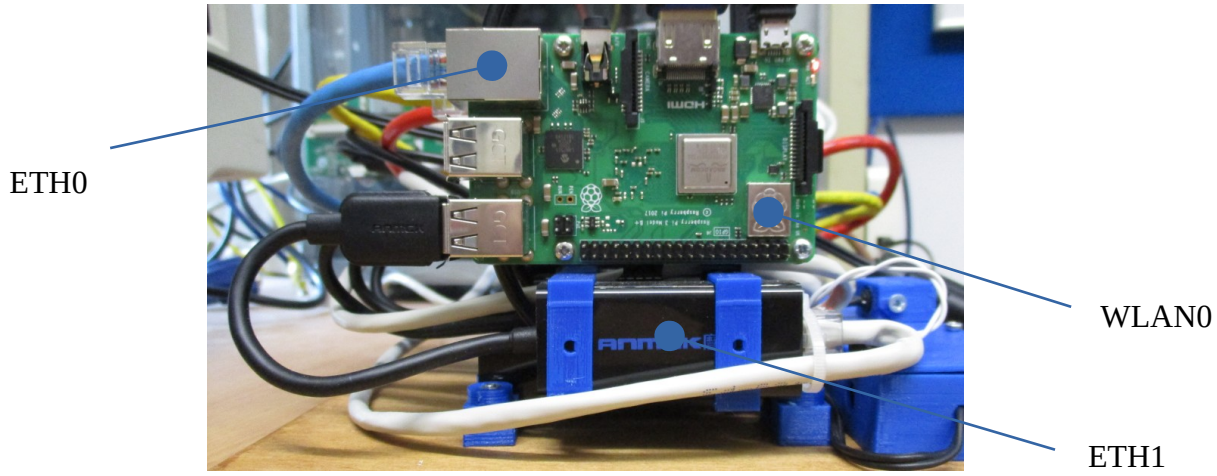


Figure 11 : Network interfaces

To start capturing network packets click on the Start icon  on the top toolbar. You will now see a lot of activity in the main window as Wireshark captures all the packets associated with the

remote desktop VNC session, as shown in figure 12. Then, click on the Stop icon on the top toolbar to stop capturing network traffic.

Note, when you first look at the traffic on a network there is a surprisingly large amount of "stuff" going on, therefore, one of the key skills in using Wireshark is filtering out the rubbish so that you can see the packets you are interested in.

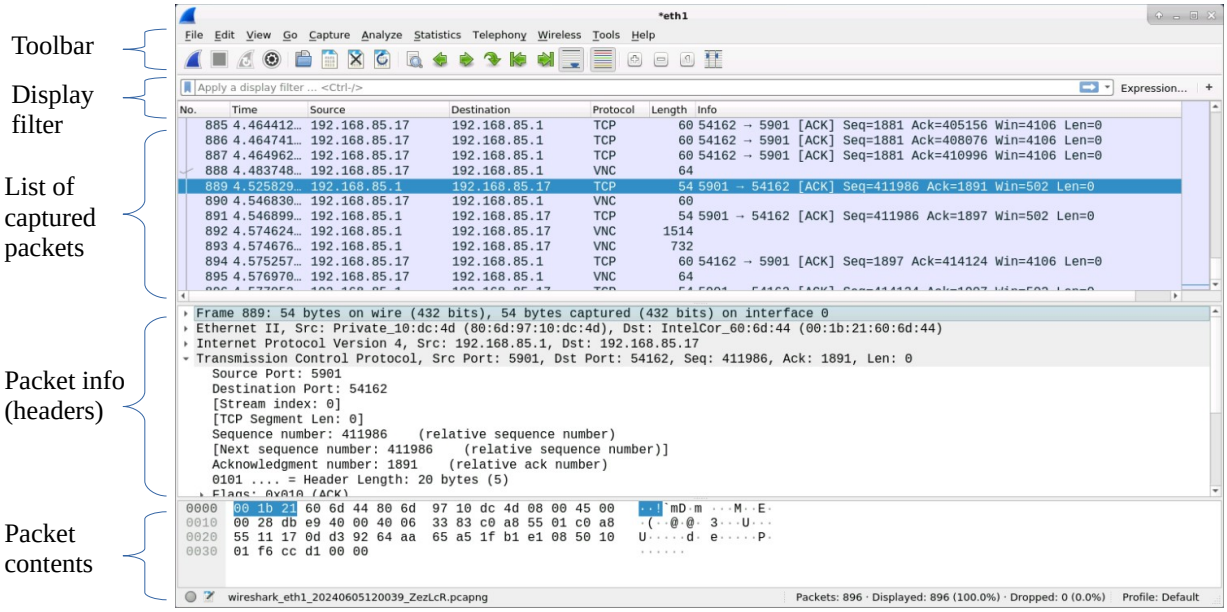


Figure 12 : Wireshark packet capture

The captured packet panel has a number of columns, displaying useful information about the captured packet, as shown in figure 13.

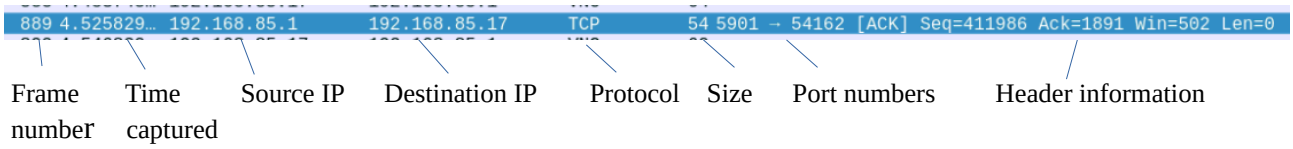


Figure 13 : Listed packet information

The packet/frame number is just a counter (ID) starting from 0 that is incremented each time a packet is captured. Time captured is the relative time from the start of the session i.e. packet number 0 is time 0.00, the difference between two packet times is their delta e.g. the time between a request and a response. The remaining columns are as labelled.

Note, you can remove or add additional columns as needed by clicking on the edit preferences icon or by clicking on the pull-down menu:

```
Edit -> Preferences
```

Then select the columns side tab, as shown in figure 14. You can then add / remove columns as needed.

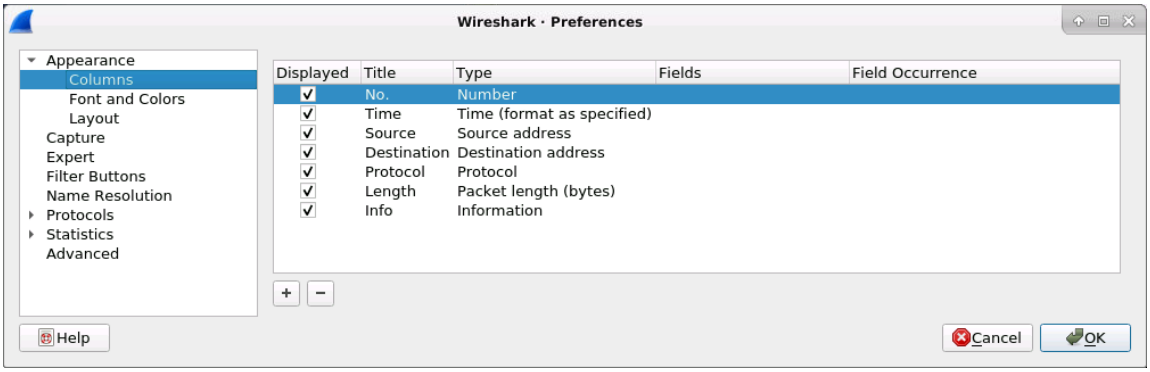
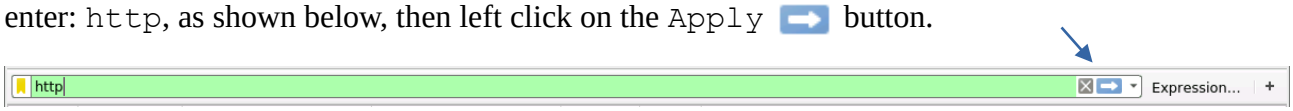


Figure 14 : adding removing columns

You can add display filters in the Filter text box shown in figure 12. To display only HTTP packets enter: `http`, as shown below, then left click on the `Apply` button.



To generate HTTP network packets, on the PC start your preferred browser and enter the IP address of your Pi-1 i.e. the URL below.

`http://192.168.X.1` `(X=Box/Desk)`

If all is connected correctly you will see Bob's web page in the browser and the associated HTTP packets captured in Wireshark as shown in figure 15 i.e. request-response packet pairs for each HTML object e.g. the image of Bob and the web-page's text.

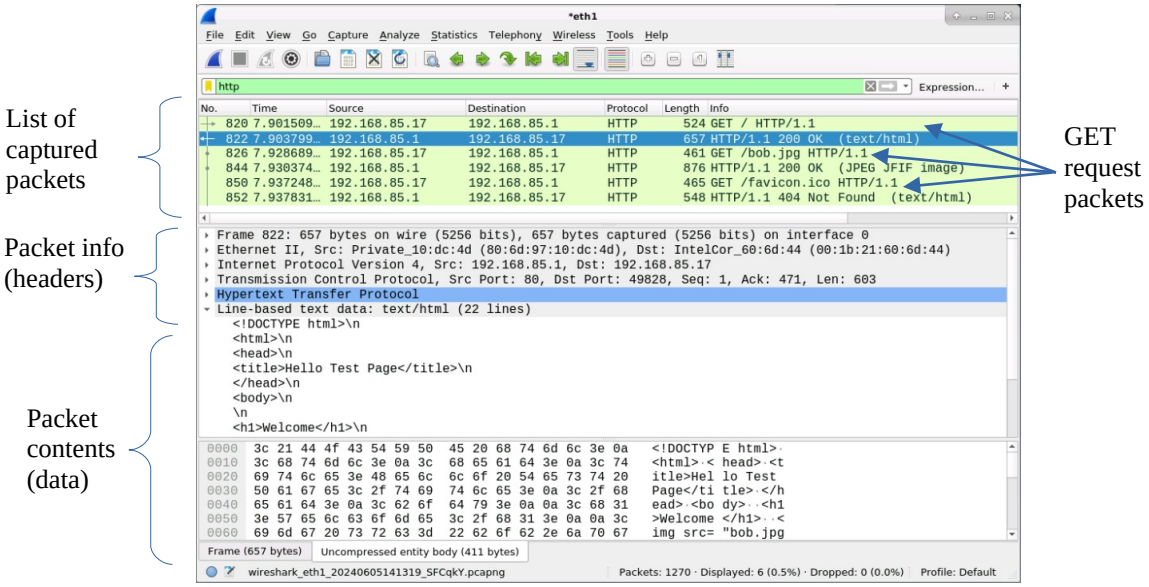


Figure 15: HTTP GET

Click on the Stop icon on the top toolbar to stop capturing network traffic.

Note, if you only see one request-response packet pair i.e. no request for `bob.jpg`, this is due to HTML caching within the browser. This is an optimisation to reduce network traffic i.e. there is no point downloading an image if its already in the local cache. To force the server to resend this data, within the browser press :

`CTRL+SHIFT+R`

Task : if all is working correctly you should have captured three request-response packet pairs i.e. you should see three `GET` requests i.e. two successful requests returning the status code 200 and one failed request, returning the status code 404, as no favicon is defined for this web page. Can you identify these packets in Wireshark?

Note, in addition to filtering on protocol type you can also filter on most of the columns shown in figure 13, some examples:

```
frame.number==100              #display the 100th frame captured
ip.addr==192.168.101.1         #display any packets with IP
ip.src_host==192.168.101.1     #display packets coming from IP
ip.dst_host==192.168.101.1     #display packets going to IP
```

You can also expand these filters using Boolean or relational functions:

```
frame.number==100 or frame.number==101  #display frames 100,101
frame.number>=100 and frame.number<110  #display frames 100-109
```

Within Wireshark you can now explore the complete Internet protocol stack, from Ethernet, IP, TCP, HTTP to the raw HTML text data used by the browser. We shall be looking at all of these protocols in later practicals. However, for the present we will focus on the HyperText Transfer Protocol (HTTP) and text data entries displayed in the packet info section of Wireshark.

Task : left click on (highlight) the first **response** packet i.e. for the `GET /` request. This will update the packet info panel, expand the HTTP section to view the request command string and headers. Then expand the Data section to see the HTML, as shown in figure 15. Can you see the same HTML text as you accessed using the Telnet command in figure 6.

Note, within the HTTP section you can left click on each header e.g. examine when the file `index.html` was last updated etc. This will also highlight the selected data / text within the packet contents panel.

Restart capturing network packets by clicking on the Start icon  on the top toolbar. You will be promoted to save the previous sessions, click on Continue without Saving. In the PC browser enter your first and last name in the two text boxes and click the submit button. This will cause the browser to generate an HTTP POST request, passing your name to the server, that in turn will return your “fortune” from the all knowing cow, as shown in figure 16.

Click on the Stop icon  on the top toolbar to stop capturing network traffic.

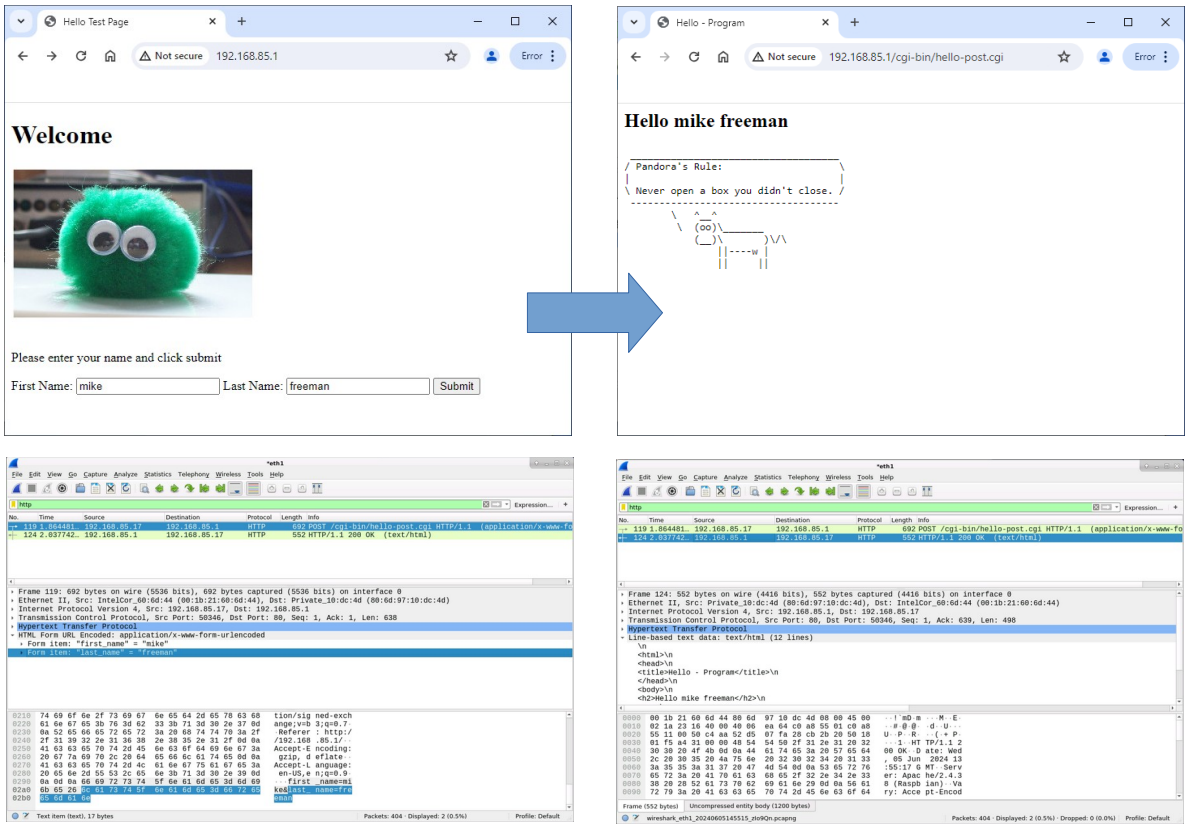


Figure 16: HTTP post (left), response (right)

Task : examine the request packet sent from the browser can you see your name within the payload section? Examine the response packet sent from the server can you see your fortune?

You can save a packet traces captured in Wireshark to a file:

`Files -> Save`

this data is saved as a `pcapng` file on the Pi, which can then be transferred to the PC as described in lab 1. You can then review / analyse this data on the PC in the lab, or your home PC in your own time. To install the Wireshark software at home follow the instructions below:

https://www.wireshark.org/download.html

IMPORTANT : do make sure you can save / use `pcapng` files. These files are required in a number of assessed questions. They also allow you to work on assessed questions outside of lab times i.e. packet captures taken in the lab can be analysed / study at home on a different machine.

Task : can you save the previous fortune packet trace to a file and open it on the PC?

Note, 30-09-2025 : sorry I forgot to ask ITS to install wireshark on the PC, I will drop them an email. However, the aim here is to make sure you can transfer the `pcapng` file to the PC so that

you can analyse these packet captures at home in your own time e.g. assessed questions :).

Task 4

Another core Internet protocol is the Simple Mail Transfer Protocol (SMTP), originally released in 1981, updated multiple times reflecting changes in usage and the internet i.e. security / encryption etc. For me this protocol was the main driver of the early Internet i.e. for better or worse the protocol that gave us email and no place to hide :).

The Simple Mail Transfer Protocol (RFC 5321: https://tools.ietf.org/html/rfc5321), port 25, is commonly referred to as a "push" protocol i.e. allowing you to pass (push) emails and attachments to a server. You can not use SMTP to read emails only send them. Unlike HTTP, SMTP is a connection based protocol i.e. the server stores state information regarding each connection / conversation. Therefore, when sending an email you will see a series of request / response packets transmitted between the client and server.

Note, SMTP also differs from HTTP in that it can be a one-to-many protocol i.e. multiple recipients. To read emails you have to use a different protocol such at Post Office Protocol (POP) or Internet Message Access Protocol (IMAP), which we will look at later.

SMTP request messages (commands):

- **HELO** : used to initiate a SMTP conversation. Client greets the server passing clients IP address. Server then responds with a HELO and its IP address. Typically this command has been replaced by **EHLO** as it allows the server to return additional parameters to the client informing it about supported services. Required to start a conversation / session.
- **MAIL FROM** : senders mailbox i.e. reverse-path e.g. MAIL FROM "desk-1@pi-1.local"
- **RCPT TO** : recipient mailbox i.e. forward-path e.g. RCPT TO "desk-2@pi-1.local"
- **DATA** : client requests to transfer data, server responds with code 354 if allowed. Client then passes header fields and message body. Final line contains a single "." to signal the end of data transfer.
- **NOOP** : no operation command, used to check server connectivity. If ok server responds "250 OK".
- **VRFY** : is used to verify whether a mailbox in the argument exists on the local host. The server response includes the user's mailbox and may include the user's full name.
- **RSET** : reset connection to the initial state i.e. removes buffers and state tables.
- **QUIT** : closes connection.

SMTP response messages return a status code and text message e.g. if a valid request was received : 250 2.0.0. Ok. Again, codes fall under these general categories:

- Positive Completion : 2XX
- Positive Intermediate : 3XX
- Transient Negative Completion : 4XX
- Permanent Negative Completion : 5XX

For more detailed explanation / examples of these code refer to :

https://en.wikipedia.org/wiki/List_of_SMTP_server_return_codes

To see SMTP in action we can again use Telnet to send commands. Telnet or `SSH` into Pi-2. Then within this new terminal window start a new Telnet session with Pi-1's mail server on port 25 by entering the following command :

```
telnet 192.168.X.1 25          (X=Box/Desk)
```

This will clear the window, allowing you to enter commands. SMTP uses a persistent connection and will automatically echo back responses. To start a conversation with the mail server at the command prompt enter each line below followed by the return ↵ key:

```
HELO pi-2.deskX.lan            (X=Box/Desk)
```

This introduces the local machine i.e. Pi-2, to the server, who in turn responds with the "command accepted" response code and its name:

```
250 pi-1.deskX.lan             (X=Box/Desk)
```

You can then send an email message to yourself my entering the following commands:

```
Client:   mail from: pi@pi-1.mail.server
Server:   250 2.1.0 Ok
Client:   rcpt to: pi@pi-1.mail.server
Server:   250 2.1.5 Ok
Client:   data
Server:   354 End data with <CR><LF>.<CR><LF>
Client:   Subject: test email
Client:   hello pi-1 from pi-2 :) <CR><LF>.<CR><LF>
Server:   250 2.0.0 Ok: queued as 1403F3EA7E
Client:   quit
Server:   221 2.0.0 Bye
```

End of DATA character sequence. Do not type <CR><LF>

Note, do NOT type "Client:" and "Server:", these just indicate Client or Server requests or responses. The text `<CR><LF>` is a carriage return sequence generated when you press the ↵ key i.e. `CR` is carriage return and `LF` is line feed. Do NOT type these, just press the ↵ key e.g. to signal the end of the `DATA` section to the server you need to press the key sequence:

↵ . ↵

Task : using the above example send an email from Pi-2 to Pi-1.

Note, an email address is defined as: user@domain. This domain name does not need to be the same as mail server's host name. They can be the same, but these are different name spaces, as illustrate in the previous examples i.e. `pi-1.mail.server` is the email domain name and the name of the host running this mail server is `pi-1.deskX.lan`.

As we are emailing ourself the email will be stored in our local mail-box, therefore, we do not need

to retrieve it from a remote server across the network. This replicates how people would sent / read emails in the early days of SMTP i.e. people would typically remote log into a “mainframe” via a terminal or telnet. To sent / read emails you needed “direct” access to a shared central computing resource that was always on i.e. ready to receive incoming mail. However, as the home computer market exploded in the 1980s this became impractical i.e. leaving your PC on, permanently connected to the Internet. Therefore, there was a need for a “pull” type protocol e.g. POP, which we shall be looking at in the next task.

However, for the moment we can replicate the “direct” access method on Pi-1 using `mutt`, via the remote desktop (VNC) as shown in figure 17, or by typing the command `mutt` in a Telnet or SSH terminal connected to Pi-1.

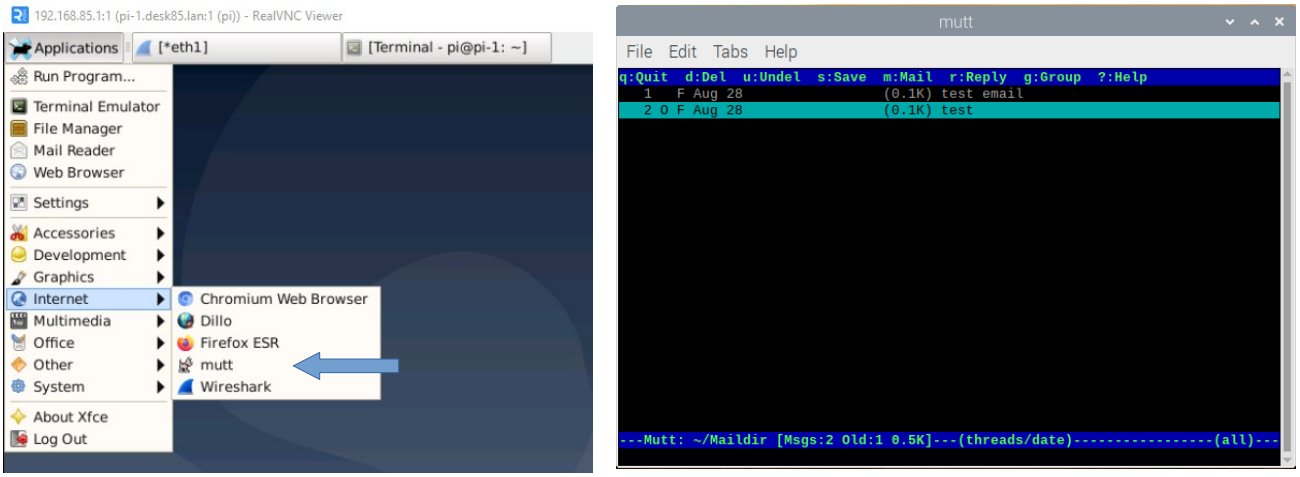


Figure 17: accessing email locally using mutt

Note, using `mutt` in this way does not use a network connection, rather you are simply accessing the files stored in your home directory on Pi-1 i.e. a mailbox in the `Maildir` folder where the email is stored.

`mutt` is a text based application, keyboard only. Use the cursor keys to select an email i.e. scroll up and down. Then to open this email press the return ↵ key. Other functionality can be selected using the keys listed in the top banner. For more information on `mutt`:

http://www.mutt.org/doc/manual/

Task : using `mutt` can you read the email you have just sent from the local mail server on Pi-1?

Task 5

To allow users remote access to their emails i.e. to remove the need for an user to be logged into the mail server, we will need another protocol. Two commonly used ones are:

- POP : Post Office Protocol - (RFC 1081: https://tools.ietf.org/html/rfc1081)
- IMAP : Internet Message Access Protocol - (RFC 3501: https://tools.ietf.org/html/rfc3501)

In general the newer IMAP protocol has replaced POP as it supports a wider range of functional, but for the purposes of demonstration we are going to stick with POP as its easier to understand :). The first version of POP was released in 1984, using port 110. Note, three year after SMTP.

Each Raspberry Pi is equipped with a POP server allowing you to read your emails from a different Pi or the PC. A selection of commonly used POP request messages (commands) :

- **USER** : user name of the connecting client (in this case pi)
- **PASS** : password of the connecting client (in this case 12345)
- **LIST** : return ID and size of emails that can be read
- **STAT** : total file space used to store emails
- **RETR <ID>** : retrieve email using its ID number e.g. retr 1
- **DELE <ID>** : delete email from server using its ID number e.g. dele 1
- **QUIT** : close connections

In the existing Pi-2 terminal start a new Telnet session with Pi-1's POP server on port 110 by entering the following command :

```
telnet 192.168.X.1 110          (X=Desk, port=110)
```

You can then retrieve the email message you sent to yourself my entering the following commands.

Note, again, do NOT type "Client:" and "Server:", these just indicate Client or Server requests or responses.

```
Client:   user pi
Server:   +OK
Client:   pass 12345
Server:   +OK Logged in.
Client:   list
Server:   +OK 1 messages:
          1 286
Client:   retr 1
Server:   Return-Path: <pi@pi-1.mail.server>
          X-Original-To: pi@pi-1.mail.server
          Delivered-To: pi@pi-1.mail.server
          Received: from pi-2 (pi-2.desk08.lan [192.168.8.2])
              by pi-1.desk8.lan (Postfix) with SMTP id 9B90141B1A
              for <pi@pi-1.mail.server>; Thu, 17 Jul 2025 11:03:38
              +0100 (BST)
          Subject: test email

          hello pi-1 from pi-2 :)
          .
Client:   quit
Server:   +OK Logging out.
```

Task : did this work? Can you read emails sent to Pi-1 from Pi-2 using POP i.e. can you access your email across the network?

Task 6

On the Internet an email is submitted using a Mail User Agent (MUA) to a Mail Submission Agent (MSA) using the SMTP protocol. Then Mail Transmission Agents (MTA) using SMTP push emails from mail sever to mail server using DNS lookups i.e. mail exchange (MX) records to identify the correct server IP addresses. When the email reaches the target mail server a mail delivery agent (MDA) stores the email to the specified inbox (file/folder).

We can't replicate this completely in the lab, but you can send an email from a Pi to the lab's mail server: `raspberrypi-mail.lan`, allowing you to send an email to someone at a different desk in CSE169. To do this a second user account has been setup on Pi-1 based on your desk number e.g. if you are working on desk 11 you can login to Pi-1 using the user name: `desk-11`, if you were at desk 2 the new user name will be: `desk-02` etc. To login as this user, on the PC click on the start button and select the command prompt.

`-> Command Prompt`

Within this new window start an SSH session on Pi-1 by entering the following command :

```
ssh desk-X@192.168.X.1              (X=Box/Desk)
```

when prompted enter the normal password: 12345. To connect to the remote mail server we will need to connect the Pi system to the labs network this is available using the **GREEN** network cable connected to the back network socket as discussed in lab 1.

IMPORTANT, to send emails to the remote mail server using `mutt` you must be logged into Pi-1 using your `desk-X` account not your `pi` account.

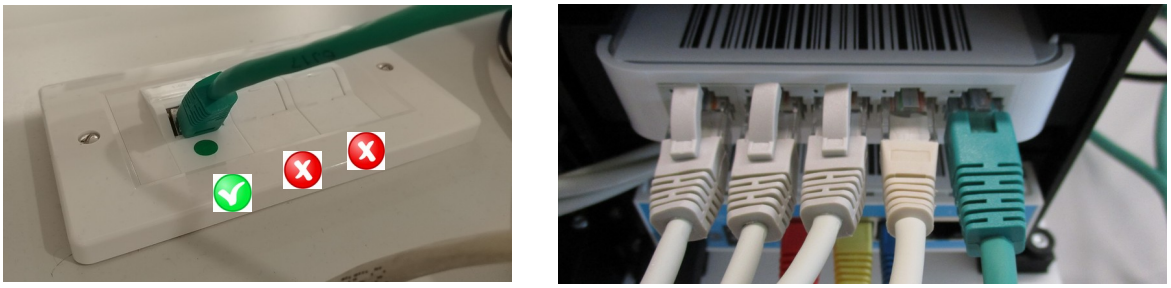


Figure 18: Lab network cable

Plug this cable into the spare socket on the network switch, this will connect you Pi system to the lab's internal network. It may take a couple of minutes for the main network switch to detect that your system is connected. To test if this connection is active, in the SSH terminal ping the labs mail server using the command:

```
ping raspberrypi-mail.lan
```

It may take a little while, but eventually you will get a response :). To send and receive emails to and from other people in the lab we can again use the mail client: `mutt`. In the SSH terminal running on Pi-1 enter the command:

```
mutt
```

This will start the `mutt` application, as shown in figure 19. This is a text based application, keyboard only, the number of emails may vary.

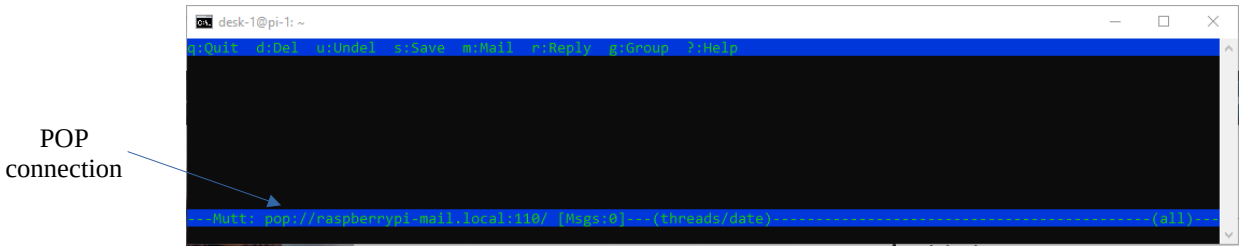


Figure 19: mutt

To send an email to another desk e.g. desk-11, press the '`m`' for mail key. A prompt will appear at the bottom of the screen enter :

```
desk-X@raspberrypi-mail.server
```

Changing desk number i.e. X, as needed, then press return ↵ key. The prompt will update to allow you to enter a subject, enter:

```
hello                          (or alternative friendly greeting)
```

`mutt` will then switch you to the nano text editor where you can enter your message enter:

```
Message from desk-X            (or alternative friendly message)
```

When you have completed your message press the `CTRL` and `X` keys to exit, then `Y` to confirm and finally press the return ↵ key to store this text to the default storage location i.e. `/tmp`. This will return you back to `mutt`.

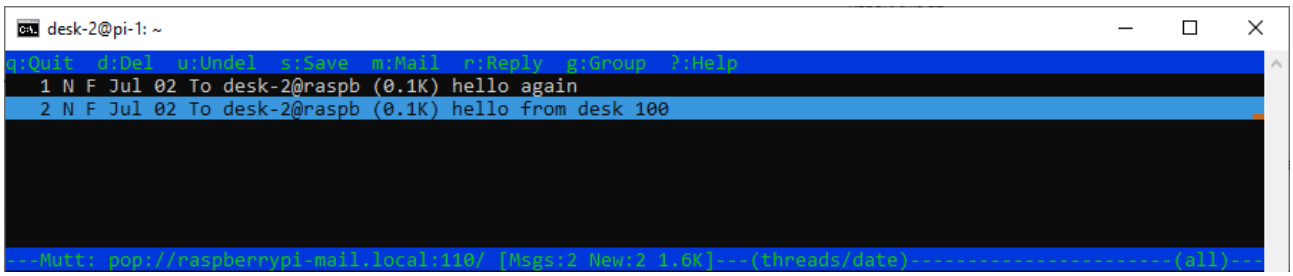


Figure 20: emails

To send this email press `Y` and if all is working and connected correctly, your email will be sent to the lab's `raspberrypi-mail` server. Emails will appear from other desks in the main window, as shown in figure 20, updating every 10 – 30 seconds. You can use the cursor keys to scroll up and down these messages, press the return ↵ key to view.

Task : can you identify how your email got from Pi-1 to the lab's mail server i.e. can you identify the Mail Transmission Agents (MTA) used? To identify the path taken open a VNC session on each Pi and run Wireshark using the filter: `smtp`. Starting with Pi-1 examine where the email is sent i.e. the destination IP address. Then repeat this process on the identified destination host.

Hint, its not a direct path, the email will take multiple hops i.e. MTAs. Also remember that the email address is different from the host name (address) , you can not use the hostname as an email address:

- Src host name (name mapped to an IP address) : `pi-1.deskXX.lan`
  - `IP = 192.168.X.1`
- Src email address : `desk-X@pi-1.mail.server`
- Dst host name (name mapped to an IP address) : `desk-X@raspberrypi-mail.lan`
  - `IP = 192.168.100.2`
- Dst email address : `desk-X@raspberrypi-mail.server`

An alternative to using VNC and a graphical packet capture is `tcpdump`. This is a command line

```
sudo tcpdump -i eth1 -n -vv -X port 25
```

`-i eth1` : listen on interface eth-1
`-n` : don't resolve hostnames
`-vv` : very verbose output
`-X` : show packet contents in both hex and ASCII

Task : make sure you are happy that you know what SMTP and POP commands are being used to send and read this email. If you are not sure, or would like to double check open an VNC to Pi-1 and run Wireshark and capture these packets. Have a look through this packet trace, see if you can identify what packets are used to send and retrieve the email.

Note, use the Wireshark filter: `smtp or pop`

When you have finished, delete any emails you have received, and close all open VNC, Telnet and SSH sessions. Then shut down the Raspberry Pi system as described in lab1, unplug and returned to its box.

Appendix A : mail server

For more information on the lab's mail server and its overheating issues :)

http://simplecpudesign.com/networking_1U_rack/index.html

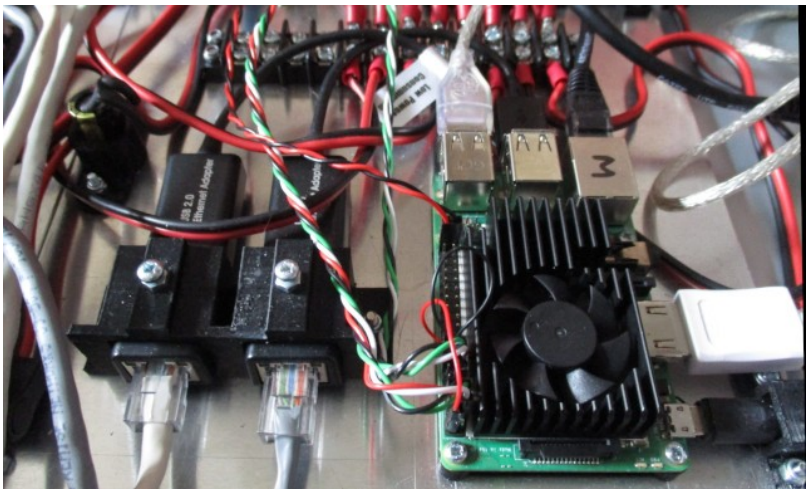


Figure 21: mail server Pi