SYS2 (NET) Laboratory 5 : TCP and IP

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 examine the protocols that for me defines the Internet: Transmission Control Protocol (TCP) and its partner the Internet Protocol (IP). Before starting this lab make sure you have watched video **Lecture 3B part 1 & 2**. These protocols are the fundamental building blocks on which the Internet was built. The first requirement of any network is reliability. Owing to electromagnetic interference if you transmit data down cables errors will occur e.g. bit flips, where 0's are changed to 1's or 1's to 0's. Therefore, we need a way to detect and retransmit damaged packets. On first consideration this may sound relatively simple as each packet has a checksum. If this does not match the received data a client "simply" needs to send a request to the server that this data is retransmitted. Unfortunately, as no part of this communications process can be guaranteed to be error free, or that packets transmitted will be received in order, or just lost, this process is actual very, very complex :(. The second requirement of any network is flow / congestion control. In a packet switching network were each host can use the full channel bandwidth you need to prevent network hogs i.e. you need a mechanism to try and prevent one machine from stealing all the bandwidth, effectively disconnecting all other hosts. TCP fulfils both of these requirement. In this lab we will start to introduce the main concepts of TCP. We will continue this investigation and look at the IP protocol in the next lab (TCP is a complex protocol). At the end of this practical you will understand how :

- TCP connections are established i.e. the three way handshake.
- To identify the different types of TCP packets e.g. SYN, ACK, PSH, RST, FIN etc.
- Reliable connections can be implemented using SEQ numbers, time-outs and retransmissions.

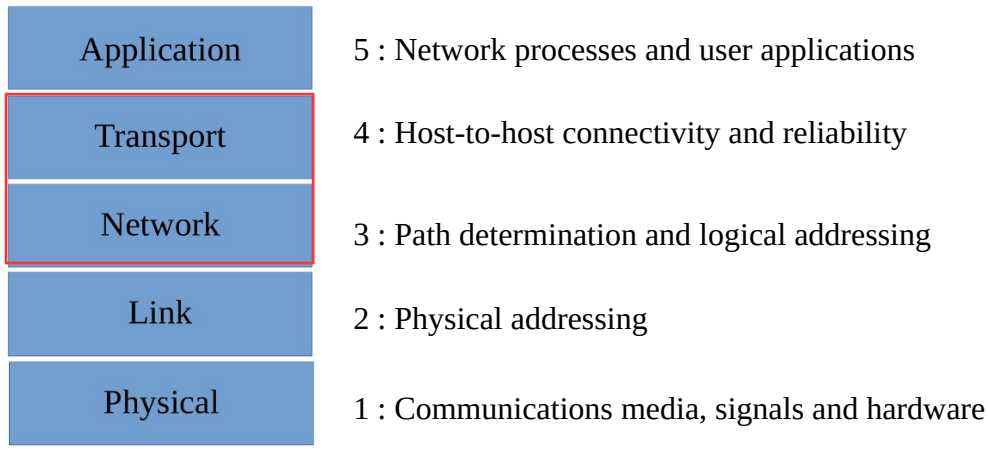


Figure 1 : The Internet protocol stack

In the last lab we looked at UDP. This is a fire and forget protocol i.e. the transmitting host does not check if the receiving host is ready, or that it received the data sent. On a small local network this is not a big issue, as the probability of data being corrupted, or that one machine will swamp the

network with packets is small. However, transmitting data to the other side of the world, through multiple countries and unknown systems is another kettle of fish. In these scenarios we need to assume that there is a significant probability that some packets will be corrupted or lost. To see TCP in action we will look at a new application layer protocol of my own creation, the cowsay protocol (CWSP) and the cowsay server.

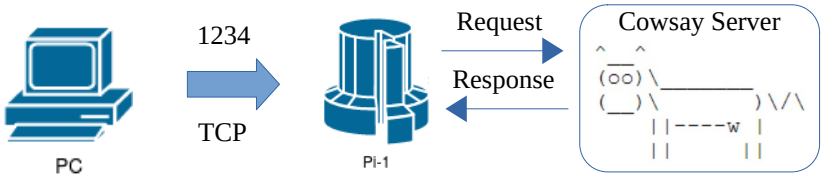


Figure 2 : Cowsay server

Text sent to port 1234 on the Raspberry Pi will be returned as cowsay text. The new cowsay protocol (CWSP) supports the following commands:

- **HELO** : used to initiate a CWSP conversation. Client greets the server by passing its IP address, or host name. Server responds with status code, hello message and additional parameters, informing the client about supported services. Required to start a conversation / session.
- **TYPE** : modifier to cowsay graphics that will be used during this sessions:
  - Borg, Dead, Greedy, Paranoid, Tired, Wired, Young, Moose.
- **TEXT** : text string that will be passed to cowsay, end of text string is signified by a carriage return.
- **QUIT** : finish cowsay session, close connection

CWSP response messages return a status code and text message e.g. when a connection is made to the server the message: `200 pi-1.local CWSP` is returned, other codes fall under the normal categories:

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

The cowsay server has been implemented using python and can be downloaded from the VLE. To test that the cowsay sever is working correctly, on the PC click on the start button and select the command prompt.

`-> Command Prompt`

Next, at the command prompt enter the following command :

```
telnet 192.168.X.1 1234          (X=Desk/Box, port=1234)
```

Note, the number 1234 at the end is the port number I assigned to the CWSP. This port number has to be greater than 1023 i.e. the system port range 0 – 1023, and less than 64K. Port 1234 does fall into the registered port range, so you could be considered it as being reserved for the VLC media

player, but I wanted 1234 more than VLC :). For more information on ports refer to:

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

Task : using Telnet test out the cowsay server by entering the CWSP protocol commands listed above. An example session is shown in figure 3.

Note, the cowsayServer is my normal high quality python :). Therefore, if you suspect that the server has crashed, is down, SSH into Pi-1 and check its status or restart using these commands:

```
sudo systemctl status cowsay.service
sudo systemctl restart cowsay.service
```

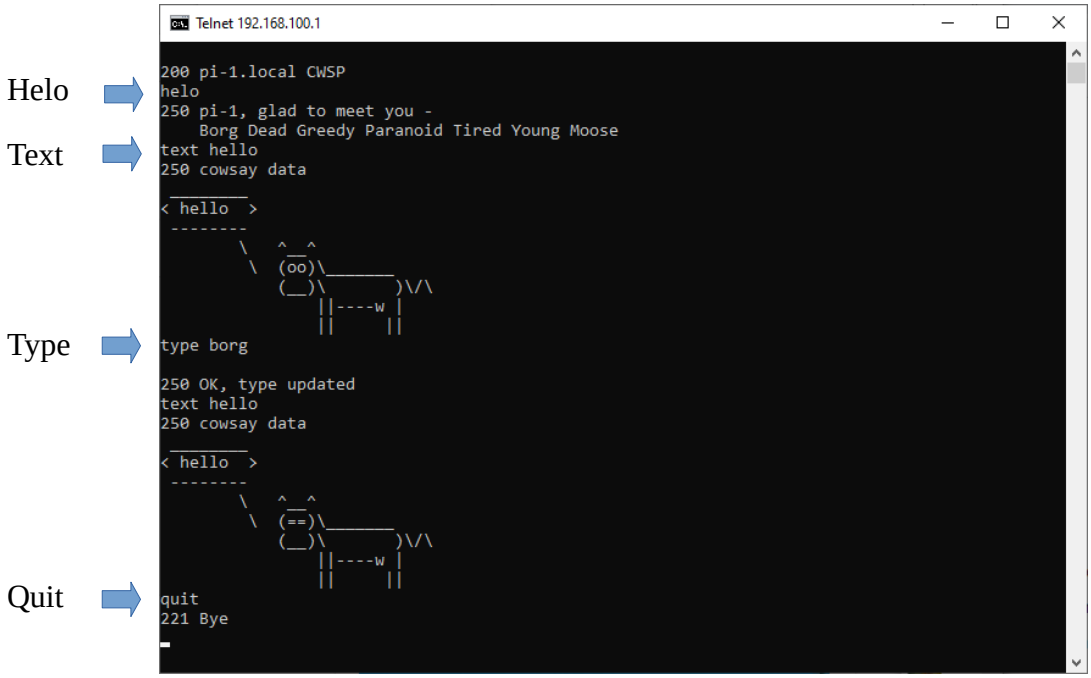


Figure 3 : Cowsay client/server dialogue

As discussed in the lecture some key points to note about the differences between TCP and UDP:

- TCP is a connection orientated protocol, whereas UDP is connectionless. What this means is that before data can be transferred the connecting client must ask the server if it is ready i.e. the TCP three way handshake. Without capturing some packets in Wireshark this is difficult to see, but you can see the difference when you compare the cowsay server to the UDP python code from the last lab:
  - UDP : `data,addr = sockRX.recvfrom(BUF_SIZE)`
  - TCP :
    ```
    sockRX.listen(1)
    sockConect, addr = sockRX.accept()
    ...
    data = sockConect.recv(BUF_SIZE)
    ```

In UDP when we wanted to RX data we call the `recvfrom()` function. Strictly speaking we should also check who the data is from (`addr`). In the previous lab we did not check this as there was only one client sending data to the server on that port. In TCP before we can RX data we have to create a connection between in the client and server using the function: `sockRX.accept()`. Once this connection is established all data is transmitted across this connection (`sockConect`). Therefore, when using TCP we can just use the `recv()` function, we do not need to specify the IP as a connection has been established between two specific hosts i.e. a point-to-point link.

- TCP is a stream orientated protocol whereas UDP is packet orientated. Again this is difficult to see from the code, however, TCP considers data to be a continuous stream of data flowing between the client and server i.e. a bi-directional link. Therefore, the TCP protocol is responsible for breaking this stream up into segments that can be transmitted across this link and reassembling them at the other end. When using UDP if this functionality is needed it is passed up to the application layer i.e. the programmer / application decides how data is broken down into segments etc. Remember UDP is designed to be a simple lightweight transport protocol, more complexity = slow. Again, you can see these differences when you compare the cowsay server to the UDP python code from the last lab:
.
  - UDP:
```
for i in range(0,number_of_segments):
    sockRX.settimeout(2)
    data,addr = sockRX.recvfrom(BUF_SIZE)
```
  - TCP:
```
data = sockConect.recv(BUF_SIZE)
```

  In last lab's UDP example we broke the data down into separate segments, whereas in TCP we do not, this is done automatically by the TCP implementation.

  Note, this example is slightly contrived i.e. the UDP segment size can be a lot bigger, but the point is still valid.

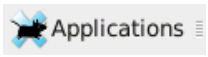
- TCP is a reliable protocol, whereas UDP is unreliable. I always feel this is a very misleading statement. UDP is not a lower "quality" protocol, it just doesn't guarantee that a packet will be delivered. On a network a UDP packet will only be "lost" if something goes very wrong on the network e.g. a damaged cable, electrical noise, or congestion (packets are dropped by a switch/router owing to excessive loading). These situation are not the fault of the protocol :). However, if these situations do occur, TCP can recover, whereas UDP can not. That is why it is labelled as unreliable.
- TCP is designed to support congestion control, whereas UDP is not. In the ideal packet switched network each host will use the network at different times i.e. be able to make use of the maximum channel bandwidth. However, as we all know having shared a single broadband connection at home this is not the case. This is due to bottlenecks in the network, for home systems this is the router. Typically for a copper connection you have a 1Mb/s upload speed from the house to your ISP. However, the connection from your PC to this router is 100Mb/s. Therefore, as the router has to store-&-forward each packet it is very easy for one host to fill the router's output buffer i.e. 100Mb/s in and only 1Mb/s out, preventing it from receiving any new packets from other hosts.

IMPORTANT, make sure you are happy connecting to and using the cowsay server as we will be

using this server in the following tasks. Over the next two labs we will be using the cowsay server and Wireshark to examine each of the previous points in more detail.

Task 1

The first step in ensuring that client data is received is to check that the server is ready to receive that data. This is achieved by the TCP three way handshake that starts all communications i.e. establishes the initial connection. To capture the network packets associated with this three way handshake open a VNC session on Pi-1. Then within the VNC session start Wireshark, click on the menu icon and select :



`-> Internet -> Wireshark`

This will open the Wireshark GUI. Next, select the network interface (NIC) to capture packets on, left click on (highlight) the Ethernet 1 (`eth1`) in the Interface list. In the Wireshark GUI enter the packet protocol filter for TCP :

```
tcp.port == 1234
```

and click on the Apply button (shown later in figure 5 if you would like to check). This will filter out all TCP packets not associated with the cowsay server i.e. port 1234.

Before we start capturing packets we need to make sure that any previous TCP connections between the PC and the cowsay server are closed i.e. not waiting for a connection. Within the VNC desktop click on the Terminal icon  . To see the current ports open on the Raspberry Pi run the following command:

```
netstat -atp
```

were:

- a = active
- t = TCP
- p = process ID

for more information on this command and its switches refer to its man page, to exit press '`q`':

```
man netstat
```

If you do not see any active connections to the cowsay server i.e. port 1234, then all is good. If you see a connection between the PC and Pi-1 is still present i.e. in a `CLOSE_WAIT` state, as shown in figure 4, then the cowsay server is not ready. To fix this problem there are two solutions:

- Wait a couple of minutes : this connection will be automatically removed after a time-out. This has probably already occurred when you were reading the man page, you can check this by running `netstat` again.
- Restart the cowsay server : in the terminal running on Pi-1 using the following command:
  - `sudo systemctl restart cowsay.service`

```
pi@pi-1:~ $ netstat -atp
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:imaps           0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:pop3s           0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:5901            0.0.0.0:*               LISTEN      888/Xtightvnc
tcp        0      0 0.0.0.0:pop3            0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:imap2           0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:x11-1           0.0.0.0:*               LISTEN      888/Xtightvnc
tcp        0      0 pi-1:1234               0.0.0.0:*               LISTEN      2984/python
tcp        0      0 0.0.0.0:ftp             0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:ssh             0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:telnet          0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:smtp            0.0.0.0:*               LISTEN      -
tcp        3      0 pi-1:1234               pc:58059                CLOSE_WAIT  2984/python
tcp        0      0 pi-1:5901               pc:51602                ESTABLISHED 888/Xtightvnc
tcp6       0      0 [::]:http               [::]:*                  LISTEN      -
tcp6       0      0 [::]:ssh                [::]:*                  LISTEN      -
pi@pi-1:~ $
```

Figure 4 : active ports on Raspberry Pi

When this old connection has been removed, click on the Start icon on the top toolbar in Wireshark. Go back to the command prompt on the PC and connect to the cowsay server:

```
telnet 192.168.X.1 1234          (IP address will vary, X=Desk)
```

You will now see some TCP activity in the main window as Wireshark captures packets associated with this initial connection, as shown in figure 5. Finally, click on the Stop icon on the top toolbar to stop capturing network traffic. To exit out of the cowsay server cleanly enter the commands:

```
helo
quit
```

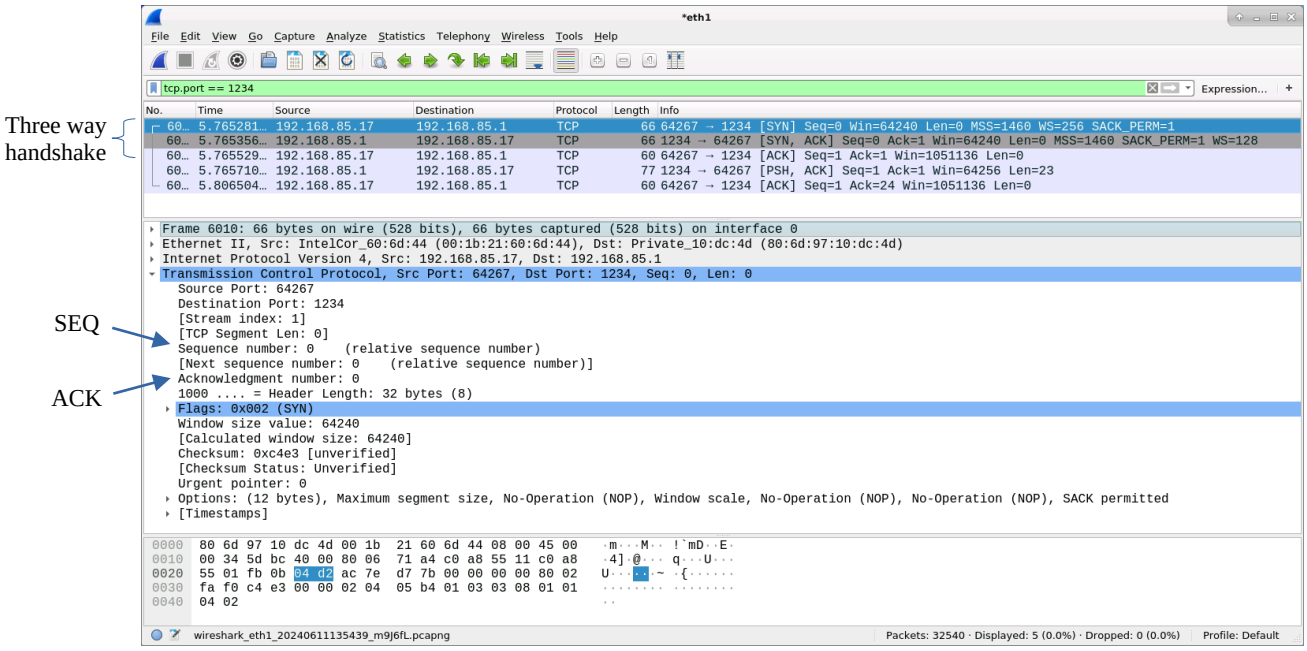


Figure 5 : TCP three way handshake packets

Task : examine the captured packets can you identify the packets used to establish this connection i.e. the the three way handshake? Can you identify the packets used to transfer the initial connection message: `200 pi-1.local CWSP`? What ports are used during this connection?

There are three packets used in the TCP three way handshake are: `SYN`, `SYN-ACK` and `ACK`, shown in figure 5. Their purpose is for the client and the server to both request and confirm their connection i.e. to signal that they are ready to communicate data. A small added complexity to this process is that one or more of these packets could get lost i.e. corrupted by noise, dropped by a router etc. In these cases the transmitting host will retransmit on a time-out e.g. if no acknowledgement (ACK) is received after a specified period of time. One final twist to consider is what will happen if the packet was not lost, just delayed e.g. due to excessive traffic on one part of the network. In this scenario the transmitting host would receive multiple ACKs.

Note, implementing a protocol to deal with all these scenarios i.e. lost, delayed packets, is tricky :)

Task 2

To understand how TCP handles these error situations we need to look at the TCP protocol in a little more detail. The Transmission Control Protocol was created in 1981 and is one of the core internet protocols (RFC 793 : https://tools.ietf.org/html/rfc793). The structure of the TCP is a lot more complex and larger than UDP i.e. UDP header is 8 bytes, TCP header is 20 – 60 bytes.

To identify if a packet has been lost or was delayed, TCP uses two new fields: Sequence (SEQ) and Acknowledgement (ACK) numbers, as shown in figure 6.

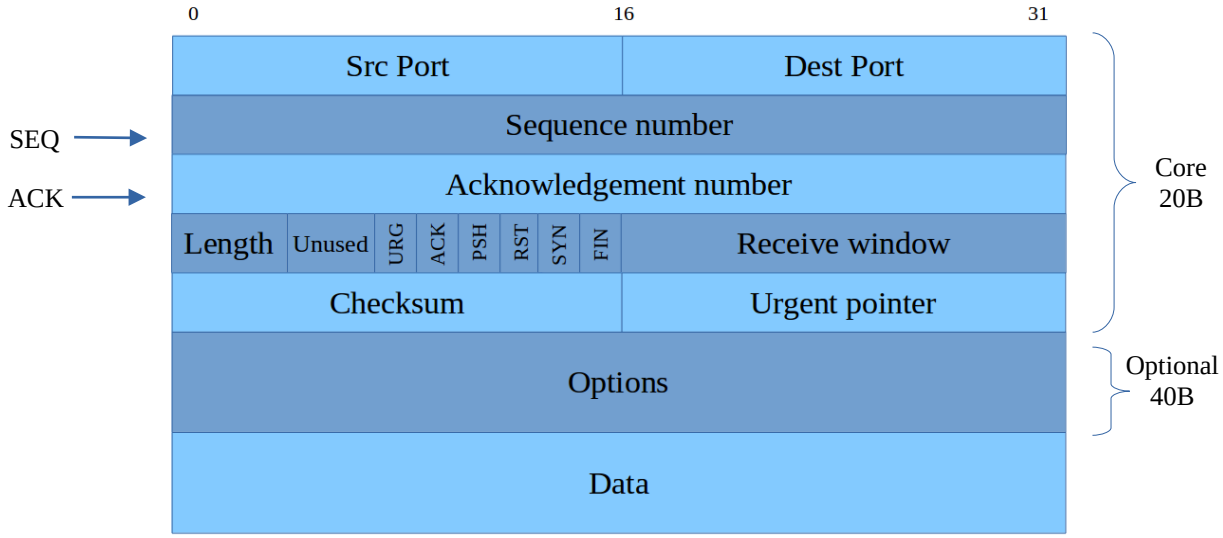


Figure 6 : TCP packet

Task : to view these numbers in Wireshark expand the TCP pull-down for each of the packets used in three way handshake e.g. the SYN packet shown in figure 5. Make sure you can identify the SEQ and ACK numbers for each packet.

Note, by default in Wireshark the SEQ and ACK values are displayed as relative numbers i.e. the SYN packet is always displayed as SEQ number 0, which simplifies things for the user. In reality

the initial SEQ number is just a random 32bit value. To view the true 32bit value you can turn this feature off by right clicking on the Sequence number or Acknowledgement number in the lower panel and deselecting (remove tick):

`Protocol Preferences -> Relative sequence numbers`

as shown in figure 7. Alternatively, this menu can be accessed from the toolbar by clicking on:

`Edit -> Preferences -> Protocols -> TCP`

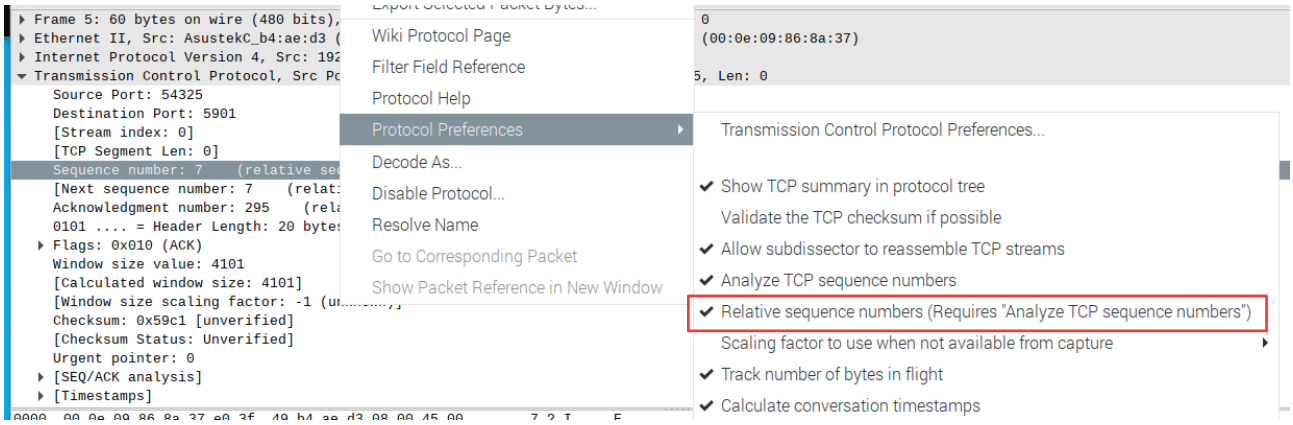


Figure 7 : enable absolute sequence numbers

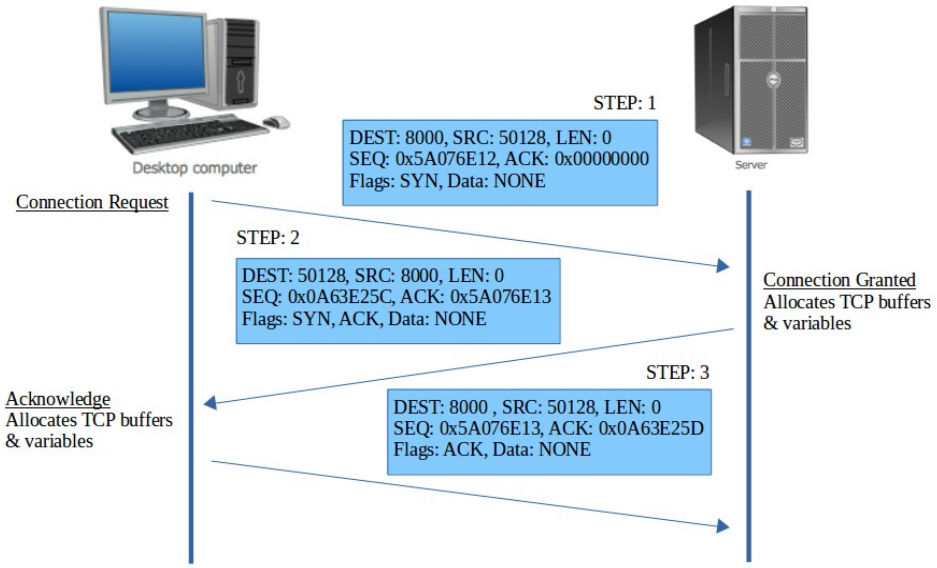


Figure 8 : sequence diagram of TCP three way handshake

To understand how these SEQ and ACK numbers are used we will have a closer look at the TCP three way handshake. To start the client requests a connection to the server using the `SYNC` packet. This is acknowledged by the server using the `SYNC-ACK` packet, which also requests a connection from the server to the client. This is acknowledged by the client using the `ACK` packet. This interaction can be represented graphically as a sequence diagram, as shown in figure 8. Sequence

diagrams contain two or more parallel vertical lines i.e. lifelines, each depicting the passage of time for each host. Time flow down these lines, starting at top, finishing at bottom. Horizontal arrows show packets being exchanged between hosts, the slope of these lines shows latency i.e. the steeper the slope the longer the delay. For more information on sequence diagram:

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

Task : to see this process in action enable absolute sequence numbers i.e. turn off relative. Then examine the `SYN`, `SYN-ACK` and `ACK` packets. Can you identify the matching values in figure 8 from your packet capture i.e. the destination and source ports, length, flags, the sequence / acknowledgement numbers and relative timings?

Note, the `SYNC-ACK` and `ACK` packets have to include information that tells the client or server what packets they are acknowledging, so that lost packets can be identified. This is done using the sequence (SEQ) and acknowledgement (ACK) numbers.

What you should see is that a packet's sequence number is used as the "ID" in its associated acknowledgement packet e.g. the acknowledgement number in the `SYN_ACK` packet is the sequence number of the previous `SYN` packet +1. This indicates to the client that the server has received all packets up to that number i.e. it is telling the client the sequence number of the next packet it expects to receive.

Note, the initial `SYN` packet's sequence number is a random 32bit value i.e. will be different for each client and server connection. These counters are incremented each time a packet is sent, or if the packet contains data by the number of bytes sent. Therefore, both client and server can use these sequence numbers to keep track of what packets have been sent and received.

Task 3

Now that we understand how a connection is established between the client and server we can next send data. To simplify this process we can replace Telnet with the python program shown in figure 9. Download this file : `cowsayServerTest.py` from the VLE. Launch a python command prompt by clicking on the start button and selecting :

`-> Anaconda`

In this command prompt change to the drive / directory containing this python program using the `cd` command. Alternatively, type "`cd` " at the command line, then within a file browser drag and drop the directory. In the Wireshark GUI ensure that the packet protocol filter is still set for TCP :

```
tcp.port == 1234
```

click on the Start icon  on the top toolbar in Wireshark. From the Anaconda prompt run the following command:

```
python cowsayServerTest.py
```

You will now see some TCP activity in the main window as Wireshark captures packets associated

with this cowsay session, as shown in figure 10. Finally, click on the Stop icon on the top toolbar to stop capturing network traffic.

Task : examine the captured packets, can you identify the packets used when transferring data i.e. from server to client (initial welcome message), from client to server (`helo` message) and the cowsay text? Do you understand how these sequence numbers are updated?

```
import socket

TCP_IP = "192.168.X.1"    # IP ADDR WILL VARY
TCP_PORT = 1234
BUF_SIZE = 1024

sockTX = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sockTX.connect((TCP_IP, TCP_PORT))

try:
  response = sockTX.recv(BUF_SIZE)
  print("RX:" + response.decode('UTF-8'))
  request = "helo"
  print("TX:" + request)
  sockTX.send(request.encode('UTF-8'))
  response = sockTX.recv(BUF_SIZE)
  print("RX:" + response.decode('UTF-8'))
  request = "text HELLO\n"
  print("TX:" + request)
  sockTX.send(request.encode('UTF-8'))
  response = sockTX.recv(BUF_SIZE)
  print("RX:" + response.decode('UTF-8'))
  request = "quit"
  print("TX:" + request)
  sockTX.send(request.encode('UTF-8'))
  response = sockTX.recv(BUF_SIZE)
  print("RX:" + response.decode('UTF-8'))
  sockTX.close()
  print("Exit")

except KeyboardInterrupt:
  sockTX.close()
  print("Exit")
```

Figure 9 : `cowsayServerTest.py` code

Note, you should have observed that the ACK flag indicates that the acknowledgement field contains the next expected sequence number i.e. a valid ACK value. Except for the initial connection this flag is always active i.e. a packet with the ACK flag set does NOT mean that this packet is acknowledging data, all it indicates is that the ACK value is valid. Equally, the PSH flag tells the client/server to use data in RX buffer e.g. end of a data transmission. This flag may NOT be set if a packet contains data i.e. typically if you are streaming data only the last packet will have the PSH flag set. In Wireshark use the `length` field to identify data packets e.g. 1500B packets.

Task : make sure you can identify what packets are acknowledgement packets and what packets contain data.

Hints, look at the packet size, in general ACK packets will have a data length of 0. Packets containing data will have a data length >0 :).

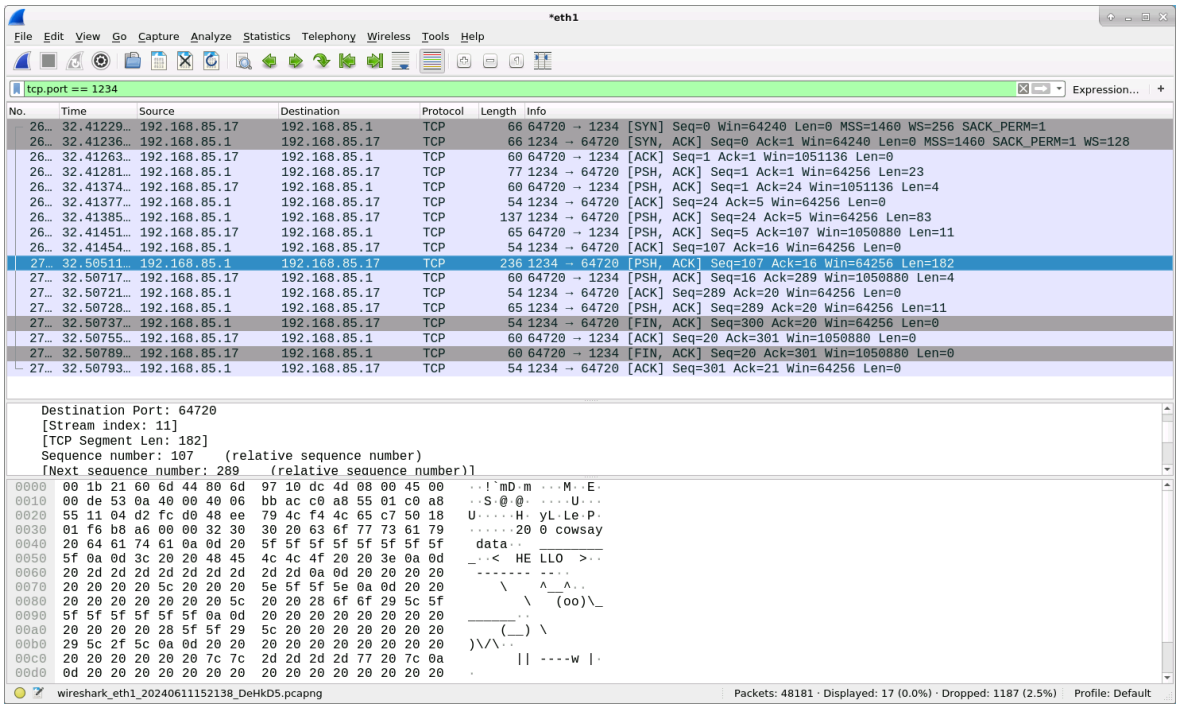


Figure 10 : cowsayServerTest code packet trace

Client					Server				
SEQ	ACK	LEN	FLG	DATA	SEQ	ACK	LEN	FLG	DATA
0	0	0	SYN						
					0	1	0	SYN, ACK	
1	1	0	ACK						
					1	1	23	PSH, ACK	
1	24	4	PSH, ACK						
					24	5	0	ACK	
					24	5	83	PSH, ACK	
5	107	11	PSH, ACK						
					107	16	0	ACK	
					107	16	182	PSH, ACK	
16	289	4	PSH, ACK						
					289	20	0	ACK	
					289	20	11	PSH, ACK	
20	300	0	FIN, ACK						
					301	21	0	ACK	

Figure 11 : packet sequence table (relative SEQ/ACK numbers).

Personally, I find reading the sequence and acknowledgement numbers used in these packet traces a bit confusing as there are two SEQ and two ACK numbers i.e. client and server SEQ/ACK numbers. Therefore, in these situations I tend to write this information out as a table, like that shown in figure 11 i.e. to help organise / keep track of these numbers, describing the packet trace shown in figure 10.

Task : can you match the packets shown in figure 11 to those captured by Wireshark in figure 10? Do you understand how these numbers are changing i.e. the link between bytes sent and ACK?

Note, a key thing to remember is that the clients / servers sequence number is simply the current sequence count + the length of the data packet transmitted. The FIN and SYNC packets count as length 1 even though they do not contain data.

IMPORTANT : make sure you understand the SEQ/ACK values in figure 11 before proceeding.

Sequence and acknowledgement numbers allow both the client and the server to see what packets the other has received, or transmitted. These numbers combined with time-outs are the fundamental mechanisms used to detect missing packets, as each host knows the packet number it should receive next. Therefore, if a packet does not arrive after a specified time i.e. the Retransmission Time-Out (RTO), or the SEQ/ACK numbers in the received packet do not match the expected values, the RX host can signal this to the TX host. To do this the RX host uses its ACK packets e.g. if the RX host is expecting to receive a packet starting with the SEQ number 100 it will repeatedly send the ACK number 100 until this missing packet is received i.e. is retransmitted.

Note, retransmissions can also be triggered if the TCP checksum generated does not match that in the TCP header, shown in figure 6, i.e. corrupt packets are dropped by the RX host. We will be looking at the role of SEQ and ACK numbers in a lot more detail in the next lab.

IMPORTANT : remember the ACK number tells the TX host what SEQ number the RX host wants next.

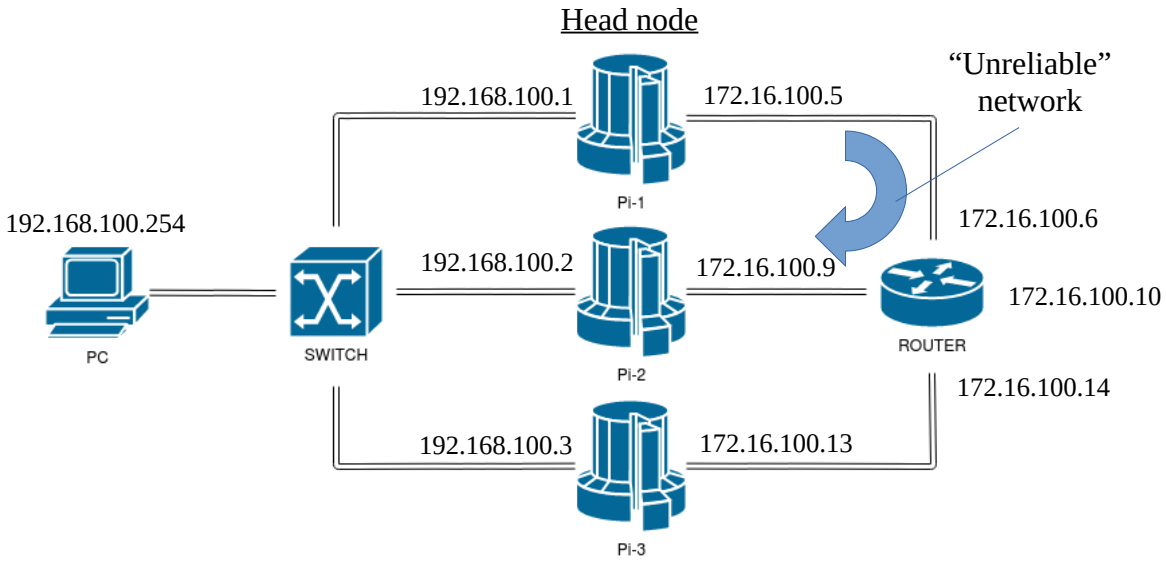


Figure 12 : example Raspberry Pi system

Task 4

Now for the tricky bit :). To see how SEQ and ACK numbers are used to detect lost packets we need to replicate the behaviour of a “noisy” network, a network where packets could be corrupted, or dropped by a router. To state the obvious we don’t want to affect the network connections running our SSH or VNC sessions, only the one running the cowsay server.

The Raspberry Pi system we are using is shown in figure 12. The network `192.168.X.Y` is used to control / configure each Pi. Therefore, we can not use this network to run these tests. However, the `172.16.X.Y` network is “free”. Therefore, we will access the cowsay server running on Pi-2 from Pi-1 across the `172.16.X.Y` network and in software we will tell Pi-2 to delay or drop packets to simulate an unreliable network. On the PC click on the start button and select the command prompt.

`-> Command Prompt`

Next, at the command prompt enter the following command :

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

This will open an SSH terminal on Pi-1. On the PC open a second command prompt and enter the following command :

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

You should now have two SSH session one connected to Pi-1 and the other to Pi-2. To control a network interface’s performance we can use the traffic control command: `tc`. For more information on the command in the Pi-2 SSH terminal enter the command below, to exit press ‘`q`’:

```
man tc
```

This command can be used to increase network latency i.e. increase the round trip delay.

Task : get a baseline of the current round trip time, in the Pi-1 SSH terminal ping Pi-2 network interface `172.16.X.9`. What is your system’s the current round trip time?

Note, as this ping only takes one hop i.e. passes through the MikroTik router, the delay should be very small i.e. a few ms.

To add 2500ms latency (2.5s) to network interface `eth0` on Pi-2 i.e. `172.16.X.9`, enter the following command in the Pi-2 SSH terminal:

```
sudo tc qdisc add dev eth0 root netem delay 2500ms
```

Task : confirm this command is working correctly. In the Pi-1 SSH terminal ping Pi-2 network interface `172.16.X.9`. What is the new round trip delay?

Restart Wireshark, this time selecting the Ethernet 0 (`eth0`) icon 

in the Interface list i.e. the

`172.16.X.Y` network. Finally enter the packet protocol filter for TCP :

```
tcp.port == 1234
```

click on the Start icon on the top toolbar in Wireshark. In the Pi-1 terminal run the following command:

```
telnet 172.16.X.9 1234          (X=Desk/Box, port=1234)
```

To generate some network traffic repeat a cowsay example from figure 3 and quit. You will see TCP activity in the main window as Wireshark captures packets associated with this cowsay session, as shown in figure 13. Finally, click on the Stop icon on the top toolbar to stop capturing network traffic.

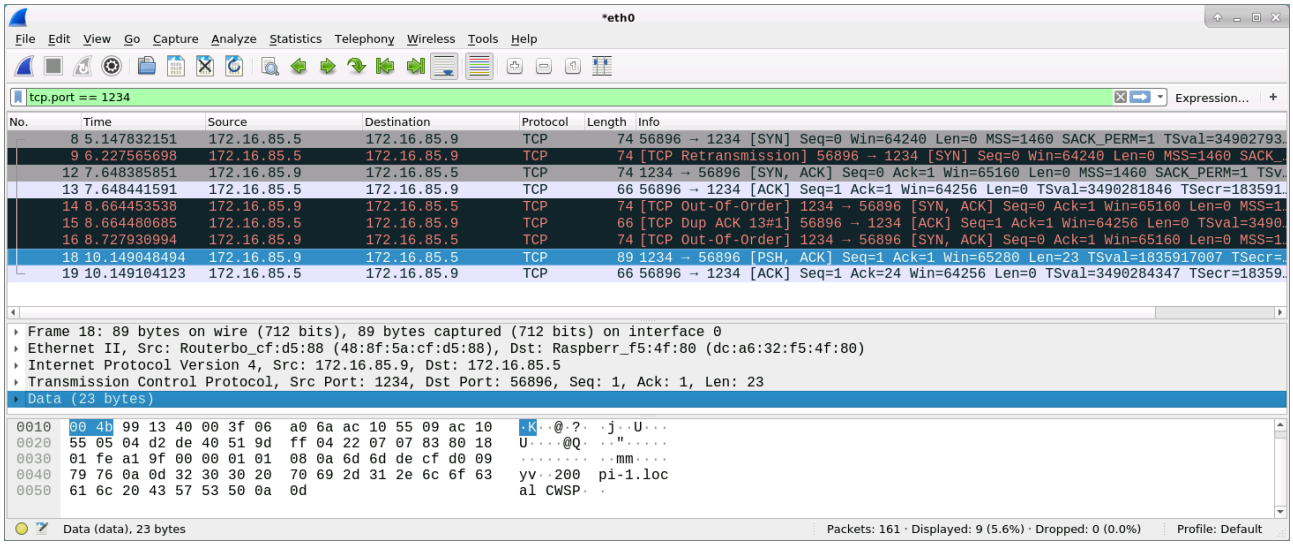


Figure 13 : Out of order (OoO) and duplicate (DUP) packets

IMPORTANT, capturing network packets is not an exact science i.e. a lot depends on what traffic has already been transferred across this socket, delays associated with the OS i.e. when are packets processed by the processor and the effects of the RTT on TCP behaviour etc. If your packet trace differs from that shown in figure 13 remove the 2500ms latency (2.5s) on Pi-2 by entering the following command in the Pi-2 SSH terminal:

```
sudo tc qdisc del dev eth0 root
```

Then test that the cowsay server is working correctly by sending the `helo` and `quit` commands, re-add the delay and try again. If all else fails you can always do a hard reset by rebooting the system :).

Note, in Wireshark packets associated with error conditions are highlighted in **BLACK**, with **RED** text. In the packet trace shown in figure 13 Pi-1 (client) is 172.16.85.5 and Pi-2 (server) is 172.16.85.9.

The sequence of packets transferred in figure 13 are shown in the table in figure 14. This example is a good example of how even the most “simple” examples can get complex really quickly :). As discussed in the lecture the RTO is dynamically generated and is based on the observed RTT between the client and the server. Therefore, when the connection is first established both the client and server have to make as best guess of what the RTO will be, the default is typically 1 second. Then as more packets are transferred each host can get a better idea of what this delay is by observing the RTT.

Task : examine your Wireshark packet capture of figures 13 and 14. Using your packet capture draw its sequence diagram i.e. the timeline of packet transfers between the client and the server. Do you understand what this diagram is showing you? Where does the RTOs occur? How does TCP handle these delayed packets?

	Client				Server				
	SEQ	ACK	LEN	FLG	SEQ	ACK	LEN	FLG	
	0	0	0	SYN					
RTO	0	0	0	SYN					
					0	1	0	SYN, ACK	
	1	1	0	ACK					
					0	1	0	SYN, ACK	OoO
DUP	1	1	0	ACK					
					0	1	0	SYN, ACK	OoO
					1	1	23	PSH, ACK	
	1	24	0	ACK					

Figure 14 : packet sequence table.

To help answer the question of how does the RTO change over time we can look at a range of statistics for this socket using the `ss` command line tool. SSH into Pi-2, then run the command :

```
ss -i
```

This will display a list of the open TCP connections and a summary of their statistics, as shown in figure 15, the key section is highlighted in **RED**.

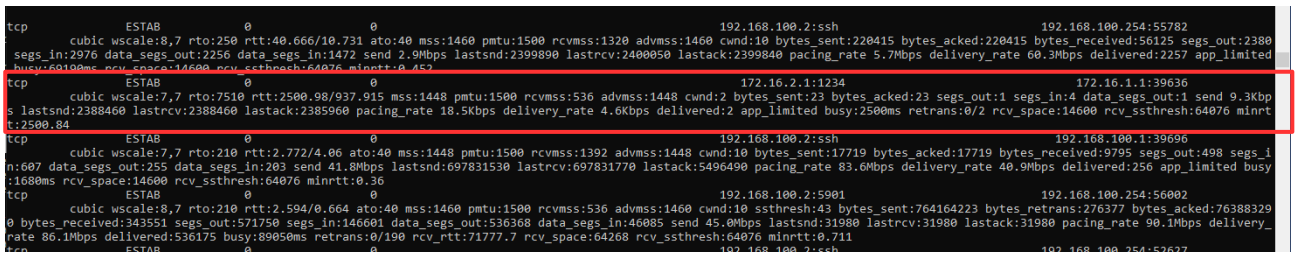


Figure 15 : socket statistics

Note, as always if you would like more information on the `ss` command examine its manual page by entering the command below into the Pi SSH terminal, to exit press '`q`' :

```
man ss
```

Task 5

In the previous tasks we looked at how a TCP connection is established i.e. the SYN packet. We also saw how data is transferred using PSH and ACK packets. The final step in the TCP communication process is to close a connection. This is done using the FIN or RST packets. The FIN packet is used to signal an anticipated connection closure i.e. a client has finished using the sever. A RST packet is used to signal that something has gone wrong and the connection is being closed owing to an unexpected problem.

Note, before performing the next section make sure you have removed the 2500ms latency (2.5s) to network interface eth0 on Pi-2, as previously described.

```
sudo tc qdisc del dev eth0 root
```

Task : to see these flags in action connect to the cowsayer server on Pi-2 again using the `172.16.X.Y` network. Using Wireshark capture the packet sequences generated for the following scenarios when using the cowsay server:

- Client closes the connection : client issues the quit command and closes the Telnet sessions
- Server closes the connection : during a client cowsay session the server unexpected terminates the connection.

can you construct the required network scenarios to generate a TCP FIN and RST flags?

Hint, the FIN is easy the RST is tricky, you may need to try a different approach :).

When you have finished 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.