SYS2 (NET) Laboratory 6 : TCP & 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.

In this week's lab we continue our investigation into the TCP protocol and its partner IP. Before starting this lab make sure you have watched video **Lecture 3B part 1 & 2**. In the last lab we started to consider the requirements needed for a reliable network protocol e.g. identifying what data has been transmitted via SEQ numbers and how ACK numbers and time-outs can be used to detect missing packets. Missing packets can occur for lots of different reasons, but a problem that can occur in any electronic equipment is noise i.e. electromagnetic interference (EMI). This interference is capable of causing single event upsets (SEU) i.e. bit flips, changing 0s to 1s and vice-versa. To detect these events both TCP and UDP use checksums, allowing a host to request retransmission of corrupted packets. The second requirement of a reliable network protocol is congestion and flow control. In a packet switching network where 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. We also need to ensure that a transmitting host does not overwhelm another by sending data faster than it can be processed. In this lab we will continue our investigation into TCP and examine how it has been designed to maximise network utilisation i.e. the different optimisations that are typically used. At the end of this practical you will understand how :

- TCP automatically breaks down larger blocks of data into MSS sized "chucks".
- SEQ and ACK numbers are used to ensure data is transferred correctly.
- to use tcptrace graphs to analyse client and server side connections.
- TCP checksums are used to detect data errors.
- the Receive window is used to control data transfer speeds.

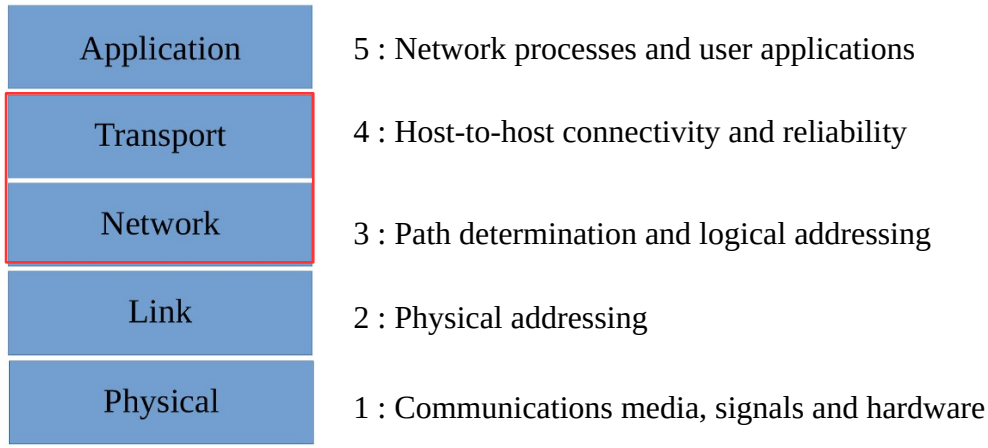


Figure 1 : The Internet protocol stack

Task 1

In the last lab we used the cowsay server as our test bench, however, we only sent very short messages that could fit into a single packet. To allow hosts to transmit larger amounts of data our protocol stack will split our data stream across multiple packets.

Task : examine the TCP and UDP packets shown in figures 2 and 3. What fields within these packets can be used to determine the size of the data sent?

Hint, if you think this is an easy question, you may want to look at the RFCs for these protocols, as Length doesn't always mean Length :). Remember TCP is a stream based protocol, whereas UDP is packet (segment) based.

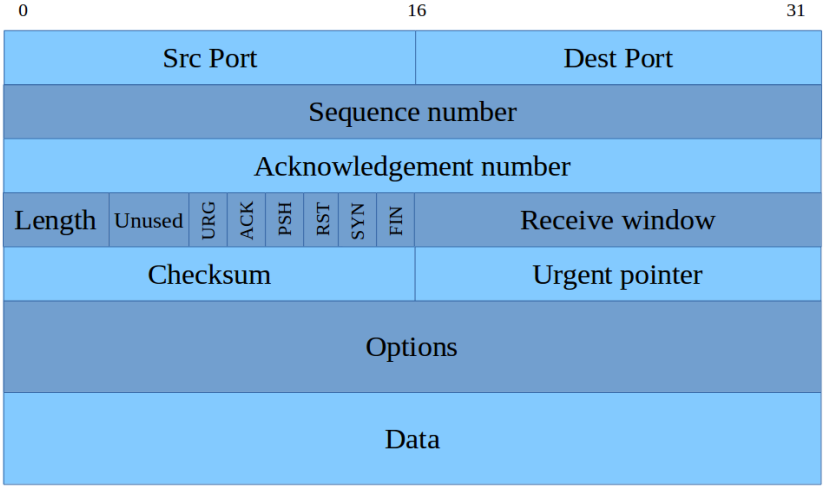


Figure 2 : TCP packet

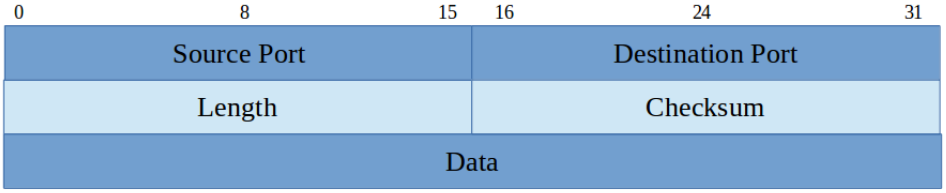
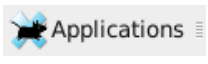


Figure 3 : UDP segment

To help understand this difference we will be using the same network configuration as the previous lab i.e. to simulate network latency we will again be using the `tc` command on the `172.16.X.2` network between Pi-1 and Pi-2. 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. Select Ethernet 0 (`eth0`) in the Interface list and enter the packet protocol filter for TCP and the port number below :

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

and click on the Apply button. This will filter out all TCP packets not associated with the cowsay server i.e. port 1234.

To run the cowsay session we need to open two SSH terminals. On the PC click on the start button and select the command prompt.

```
-> Command Prompt
```

Next, at the command prompt SSH into Pi-1 by entering the following command :

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

On the PC open a second command prompt and SSH into Pi-2 by entering the following command :

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

You should now have two SSH sessions one connected to Pi-1 and the other to Pi-2, as shown in figure 4.

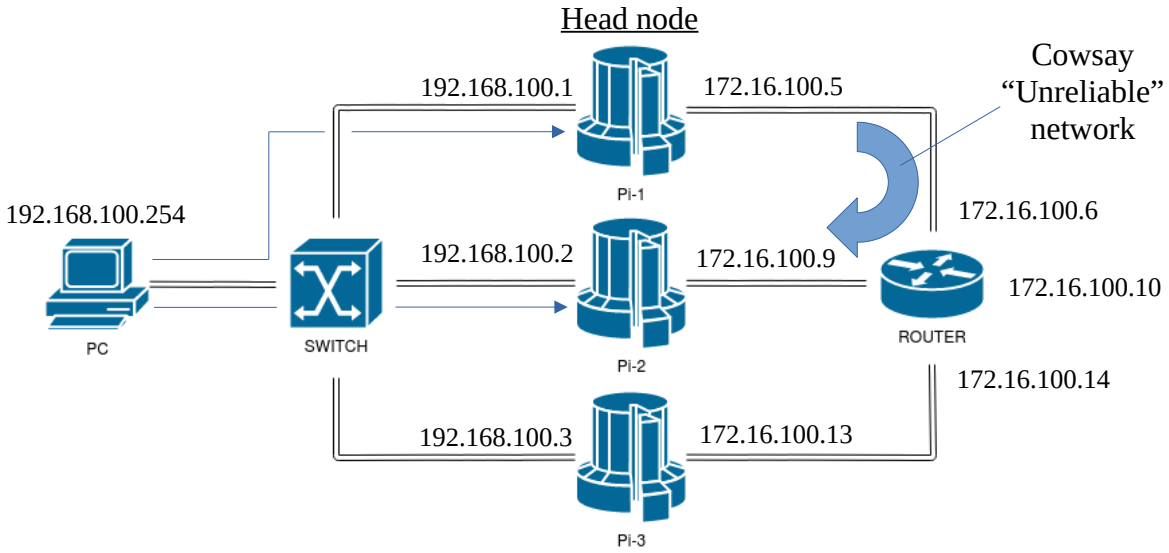


Figure 4 : example Raspberry Pi system for X=100

Download from the VLE the test program `cowsayBigTest.py` and its data file `pg164.txt`, (Twenty Thousand Leagues under the Sea) to the PC. The test program is shown in figure 5 and is designed to transmit and receive text data to and from the cowsay server running on Pi-2. The amount of data sent is defined by the `SIZE` variable i.e. by setting `SIZE` to different values we can transfer 1 to 600,000 characters.

Update the variable `TCP_IP` to the IP address of Pi-2. Copy these files to Pi-1. The python program and its data file must be stored in the same directory i.e. typically `/home/pi`.

Tip, if needed refer back to lab 4 for details on how to use the FileZilla to transfer files between the PC and a Pi.

```
1   import socket
2   import time

3   TCP_IP = "172.16.X.9"       # IP ADDR X WILL VARY
4   TCP_PORT = 1234
5   BUF_SIZE = 2048

6   SIZE = 88                   # UPDATE

7   sockTX = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8   sockTX.connect((TCP_IP, TCP_PORT))
9   sockTX.settimeout(2.0)

10  dataFile = open("pg164.txt", 'r')
11  try:
12    print( "RX:" + sockTX.recv(BUF_SIZE).decode('UTF-8') )
13    print( "TX: helo" )
14    sockTX.send( b"helo" )
15    print( "RX:" + sockTX.recv(BUF_SIZE).decode('UTF-8') )

16    data=[]
17    while (len(data) < SIZE):
18      text = dataFile.read(1)
19      if text != '':
20        if (ord(text)>0x2F and ord(text)<0x3A) or (ord(text)>0x40 and \
              ord(text)<0x5B) or (ord(text)==0x20) or (ord(text)>0x60 and \
              ord(text)<0x7B):
21          data.append( text )
22        else:
23          data.append( " " )

24    sockTX.send( b"text" )
25    sockTX.send( ''.join(str(x) for x in data).encode('UTF-8')  )
26    sockTX.send( b"\n" )

27    try:
28      while True:
29        data = sockTX.recv(BUF_SIZE)
30        if not data:
31          break

32        print(data)
33        print("### " + str(len(data)) + " ###")

34        #time.sleep(0.001)

35    except socket.timeout:
36      pass

37    sockTX.send( b"quit\n" )
38    print("RX:" + sockTX.recv(BUF_SIZE).decode('UTF-8'))
39    sockTX.close()

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

Figure 5 : `cowsayBigTest.py`

To test that everything is setup correctly we will first run `cowsayBigTest` without any network delays. In the Wireshark GUI click on the Start icon on the top toolbar in Wireshark. From the Pi-1 SSH terminal run the following command :

```
python cowsayBigTest.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 6. Finally, click on the Stop icon on the top toolbar to stop capturing network traffic.

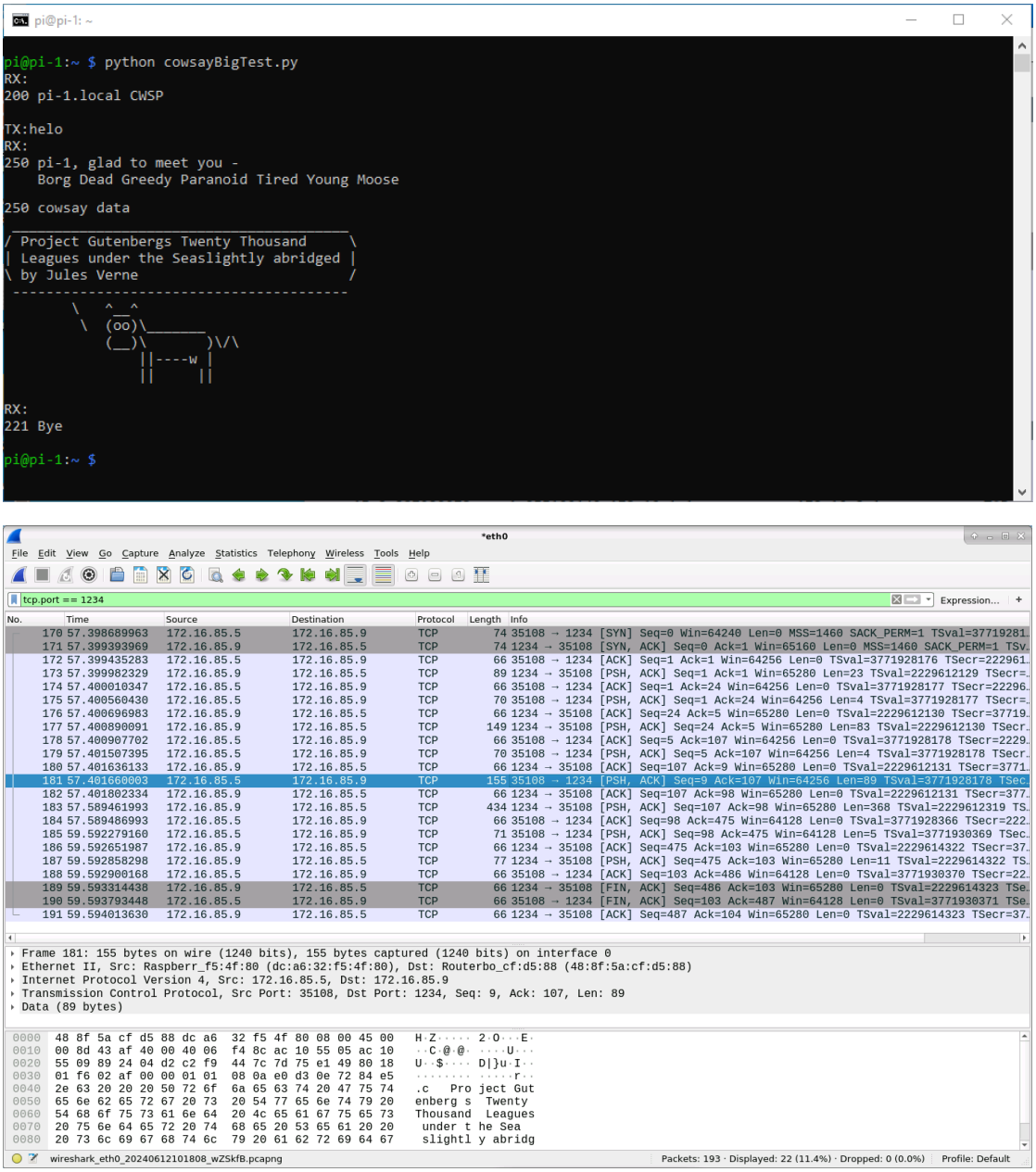


Figure 6 : `cowsayBigTest` packet trace

Note, I do confess that the `cowsayBigTest.py` program is a little messy, but it was easy to implement :). The exit condition from the receive loop is implemented using a time-out rather than a defined exit string e.g. in SMTP you signal the end of the data section with : `Enter . Enter`

IMPORTANT, this example demonstrates a typical client / server interaction, make sure you can identify each stage in the Wireshark packet trace shown in figure 6 before proceeding. Can you identify the initial 3 way handshake, the transfer of data from client-to-server, then the transfer of data from server-to-client, i.e. can you find the text that the Cow says in both directions? Then finally the closing of this connection.

Tip, in Wireshark to see if the client (`cowsayBigTest`), or the server (`cowsay`) is transmitting or receiving data look at the source and destination IP addresses. If you are not sure what these are have another look at figure 4.

Task : increase the `SIZE` variable to the values below and repeat the previous task. Each time the `cowsayBigTest.py` program is executed capture the packets transmitted using Wireshark. In each case identify how many data packets are sent? How does the `cowsayBigTest` receive buffer size (`BUF_SIZE`) affect the data packets sent?

- 2000
- 8000

Tip, to edit the python test program you can either edit it on the PC and re-transfer it to Pi-1, or you can edit this file in the SSH terminal. There are a number of command line editors available on the Pi, two I would recommend are: `nano` or `vi`. My personal preference is `vi` as its supported on all Linux systems. However, `nano` is a lot more user friendly. To edit the cowsay test program using `nano`: in the Pi-1 SSH terminal enter the command:

```
nano cowsayBigTest.py
```

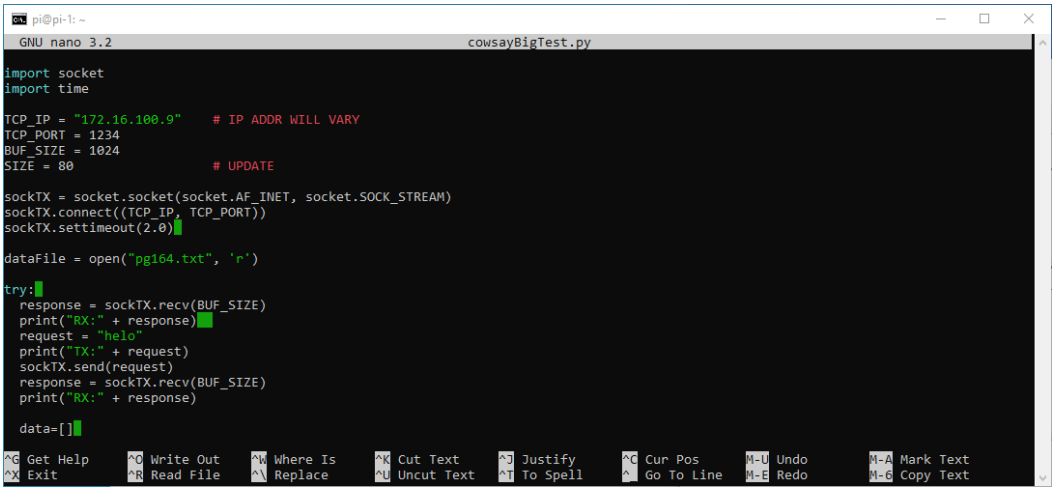


Figure 7 : nano

This will open the edit window shown in figure 7. Using the normal cursor and delete keys change

the value assigned to the variable SIZE. To save this modification press :

`CTL + s` (hold down control and s keys)

To exit this editor press :

`CTL + x` (hold down control and x keys)

This will return you to the SSH terminal.

Task 2

From the previous task we can see that the TCP protocol breaks the data transfer down into a number of network packets. To help visualise this data transfer we can examine this packet trace as a graph. Within Wireshark select a `PSH` packet, then click on the top pull-down and select:

`Statistics -> IO Graphs`

This will draw a graph of the network IO traffic for the period of time captured by Wireshark, as shown in figure 8. This may take a few seconds to draw depending on how much data you captured. This graph shows the total data transferred. If required apply filters e.g. `tcp.port==1234`, to view specific ports or TCP streams. You can then turn these plots on or off by clicking on the `enable` tick boxes.

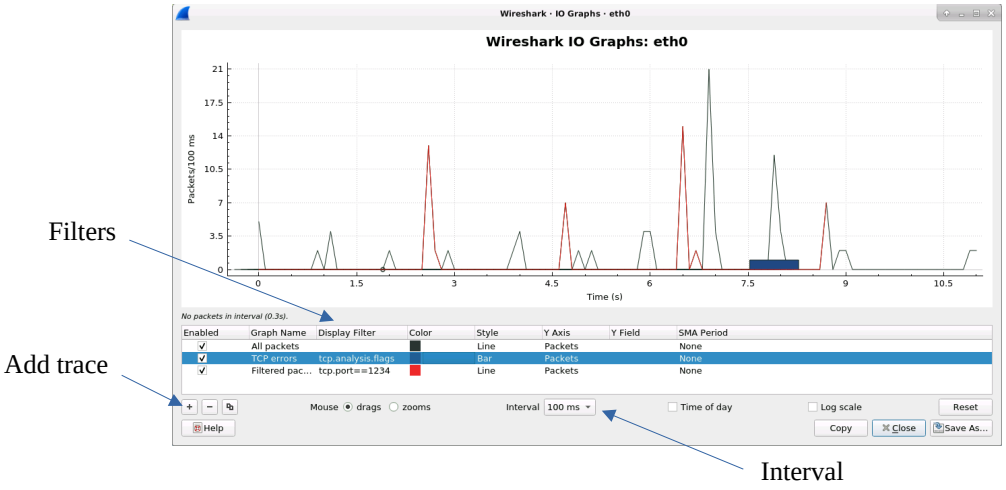


Figure 8 : IO graph

Note, you can change the graph's resolution using the Interval pull down, from 10 min to 1 ms, 100ms is a good starting point. You can move along the time axis using the scroll bar, or left click hold and drag. If you click on a peak in a graph, the associated time period will be highlighted in the main packet trace window. Finally, you can zoom in or out by right clicking on the graph.

Another useful display feature in Wireshark is the ability to follow a TCP stream. To do this, right click a data packet. You can identify these in the trace by their size (larger length) or flags (PSH), and select :

`Follow -> TCP Stream`

This will open a new window, displaying the data that was transferred across that connection, as shown in figure 9.

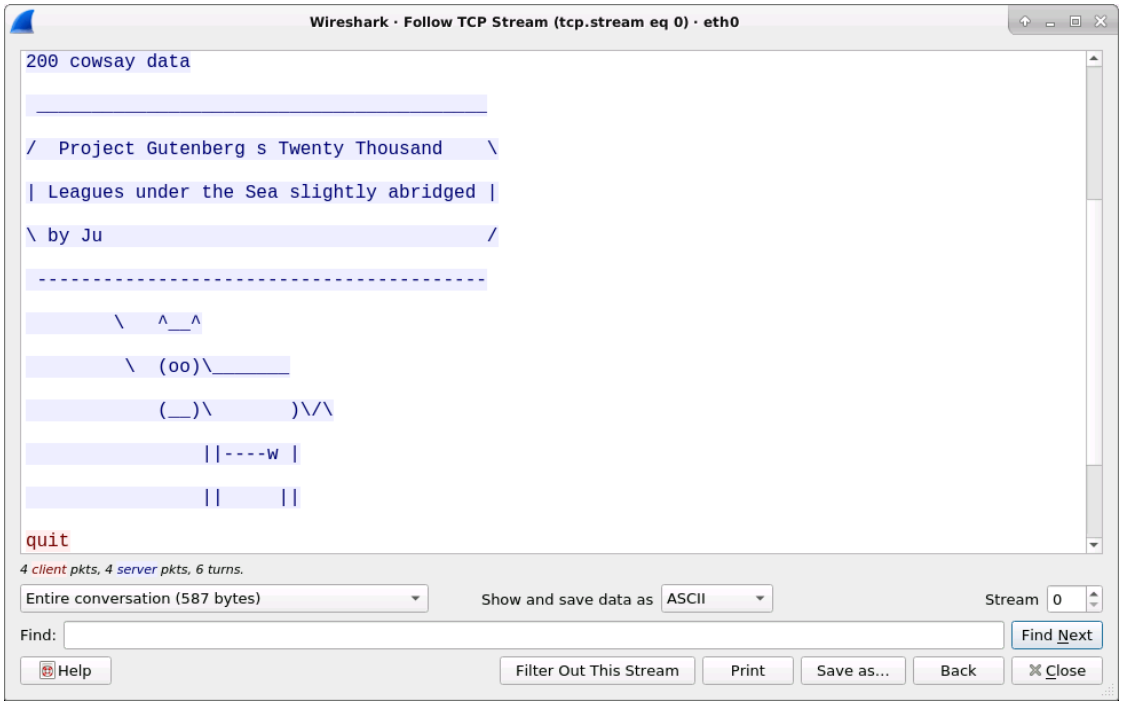


Figure 9 : TCP stream view

The default TCP data segment size is 536B. If a host wishes to increase this to improve transfer efficiency i.e. the ratio of header size to data size, this can be “announced” in the initial three way handshake i.e. a host states the Maximum data Segment Size (MSS) it will accept.

Note, the TCP MSS is dependent on the underlying protocol used in the network layer e.g. IP. These lower layer protocols define the Maximum Transmission Unit (MTU). For IP the MTU is typically 1500B, therefore :

$$MSS = MTU - TCP_{HeaderSize} - IP_{HeaderSize}$$

Task : examine the previous Wireshark packet trace, how big are the TCP and IP header i.e. in bytes? What is the MSS is used on Pi-1 i.e. how big is the data payload compared to the MTU?

To ensure that data is transferred reliably each transmitted segment must be acknowledged by the receiving host i.e. each packet is assigned a sequence number and the receiving host will send back an ACK packet containing the starting SEQ number + data length received i.e. acknowledging that it has received data correctly up to this point. Acknowledging received packets is an essential part of a reliable network protocol. However, it would be impractical to stop and wait for each ACK after a data packet is transmitted e.g. the RTT from York to America is approximately 80ms, you don’t want to wait to this amount of time each time you send a packet.

Task : examine the previous Wireshark packet trace, does the transmitting host wait for an ACK before transmitting the next data packet i.e. does it wait to see if data has been received before sending the next packet?

To help answer this question compare my packet trace shown in figure 10 to the one you have captured (they may differ a little). Clicking through these packets we can identify the first data packet sent by examining the Data (payload) section i.e. contains the starting phrase shown in figure 9, “Project Gutenberg …”. In my example: packet 26 – SRC=172.16.85.5, SEQ=9, ACK=107, LEN=1448. Using the same technique you can then identify the response from the cowsay server i.e. cowsay ASCII art text, from my example: packet 40 – SRC=172.16.85.9, SEQ=107, ACK=8010, LEN=1448.

IMPORTANT, make sure you identify these packets in your packet capture before proceeding.

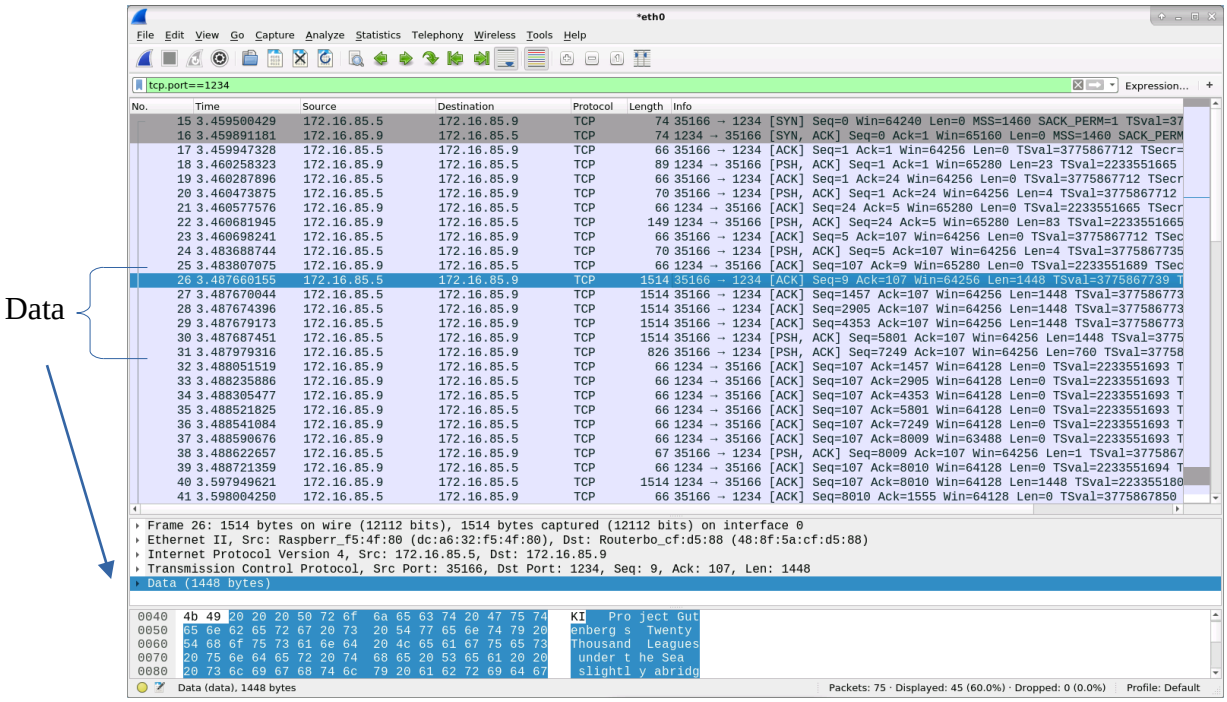


Figure 10 : wireshark packet trace, client side

Typically, to improve network efficiency the transmitting host does not wait for an ACK if it is capable of transmitting MSS sized packets i.e. if the TX buffer is full, as shown in figure 10. In this example the client has sent out 5 data packets (packets 26 - 31) before the first ACK is received back from the server (packet 32).

Note, the final packet transmitted by the client is less than the MSS, therefore, even though the PSH flag is set in this packet, it is not transmitted until the TCP transmit time-out timer has elapsed, or if there are unACKed packets, we will look at this in more detail in the next section.

Task 3

The example shown in figure 10 shows network communication from the client's point of view. To see the server's view point of this conversation we will need to open a second VNC session to Pi-2. Open a new VNC session and connect to Pi-2. Start Wireshark on Pi-2, selecting NIC eth0 and setting the filter to TCP port 1234.

Task : examine the Wireshark packet captures on the client and server side i.e. open VNC sessions on both Pi. Examine these packet captures can you identify when packets transmitted on one Pi are received on the other Pi?

We can again view these data transfers graphically in Wireshark as a TCP time sequence graph, showing how the sequence numbers change in each direction i.e. for the communications link from Pi-1 to Pi-2, and the link from Pi-2 to Pi-1.

On Pi-1's Wireshark capture select a packet that is transferring data from the Pi-1 to Pi-2 e.g. in figure 10 packet 26. You can identify a data packet in your capture by its size (1448B) and that its coming from the Pi-1 by the source IP address `172.16.X.5`. Next, click on the top pull-down and select:

`Statistics -> TCP StreamGraph -> Time-Sequence Graph (tcptrace)`

This will draw a graph of the TCP traffic for the period of time captured by Wireshark, as shown in figure 11.

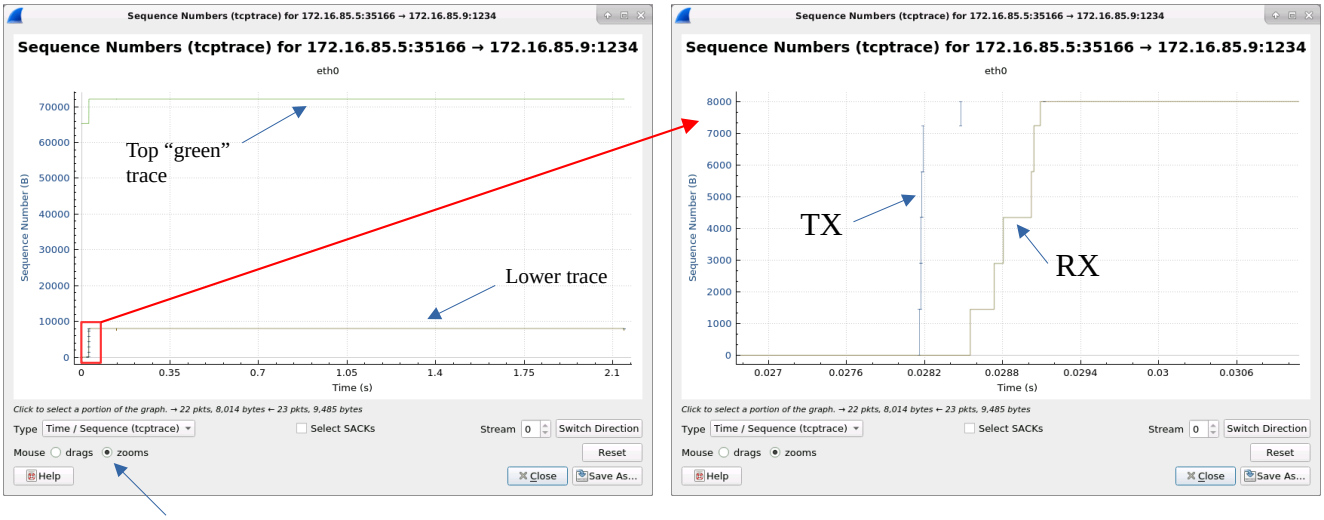


Figure 11 : wireshark tcptrace graph, initial (left), zoomed in (right)

For the moment we will ignore the top "green" trace and focus on the lower trace. To zoom in on the highlighted region change mouse mode from `drags` to `zooms`, left click, hold and drag a box around the desired area. Repeat this process until you have a graph that looks like that in figure 11.

This plot shows transmitted and received sequence numbers (bytes) against time. The blue "bars" show when data is transmitted (from client), the height of these bars is the amount of data transferred i.e. typically MSS, except for the last packet. The grey trace are the acknowledged

sequence numbers (from server). Each time an ACK packet is received there is a step change i.e. in the example shown in figure 11 we see that there are 6 data packets transmitted and 6 ACK packets received.

Task : set the `SIZE` variable used by the `cowsayBigTest` program to the maximum value of 600,000. Run this program, examine the packet capture in Wireshark as a `tcptrace` graph. Make sure you can identify when data packets are transmitted and when acknowledgement packets are received. Run this program a couple of times, do the Wireshark captures vary?

Note, another technique used to improve network efficiency are delayed ACKs. This technique tries to minimise the number of zero length ACK packets. A host is allowed to delay ACK packets for up to 500ms (varies with implementation, can be less), or tries to only acknowledge every second MSS sized packet. It can then send a cumulative ACK packet for all the data received up to that point i.e. SEQ + Total data RX. A variation on this theme is piggy-backing. Here if the receiving host needs to transmit data, rather than waiting i.e. follow the delayed ACK rules, it will immediately send this data and the ACK in the same packet e.g. a (PSH,ACK).

Task 4

The final key elements of the TCP protocol that we shall be looking at in this lab are Flow and Congestion control. Unlike UDP, when using TCP the transmitting host checks if the receiving host can keep up with the speed that data is being transmitted. For small packets this is not an issue as short duration peaks can be absorbed by the receive buffer i.e. the receiving host can finish what it is doing and process this data later from memory. However, for continuous or large data streams the transmitting host needs to make sure it is not overwhelming the receiving host i.e. is the transmitting host putting data into the receive buffer fast than the receiving host can process it. If it is this would result in inefficient network usage, as the receiving host will start to discard (drop) packets that will need to be retransmitted later. To avoid these situations the TCP protocol uses the receive window field show in figure 2 (on page 2). This field states how much data each host can buffer i.e. how much data can be sent to this host before the TX host has to wait for an ACK. Each host's window size is stated in the initial three way handshake.

Note, the receive window size effectively defines the maximum amount of data that can be in flight at any one time i.e. on the wire, buffered in routers etc. For long or high latency connections this can have a significant impact on network efficiency.

Within the TCP header the Receive Window field is a 16bit field, therefore, the maximum receive buffer size would be limited to 64K, which is a little small. Therefore, to increase this buffer's size we define a window size scaling factor in the TCP option field, as shown in figure 12. In this example the window scaling factor is 8 i.e. 2^8=256, therefore, the maximum receive buffer will be 256 times larger.

Note, to state the obvious, the receive buffer size is limited by the amount of physical memory a host has e.g. if the maximum scaling factor is 14, the maximum buffer size would be 2^14 × 2^16 = 1GB for each TCP connection, that's a lot of memory :).

Task : what is the minimum and maximum allowed receive window sizes on the Pi?

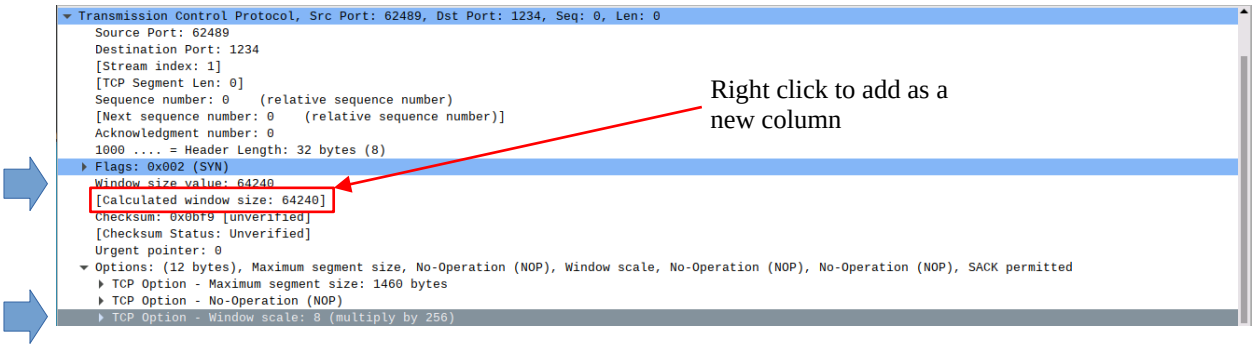


Figure 12 : Window size

The TX host can examine a RX host's receive window size i.e. the amount of free space in its receive buffer, via the receive window field within an ACK packet. To add this column to the Wireshark packet trace select a TCP SYN or ACK packet coming from Pi-1 e.g. as shown in figure 12. Then right click on the field you are interested in e.g. Calculated Window Size, this will bring up a range of options, select "Apply as Column" to add.

Set the `SIZE` variable used by the `cowsayBigTest` program to the maximum value of 600,000. Run this program, capturing its packet trace in Wireshark. You will see a lot of TCP activity in the main window as Wireshark captures packets associated with this cowsay session. As data packets are received by Pi-1 from the cowsay server you will see the value in the Calculated Window Size column decrease. You can also see this graphically in tcptrace graph as shown in figure 13.

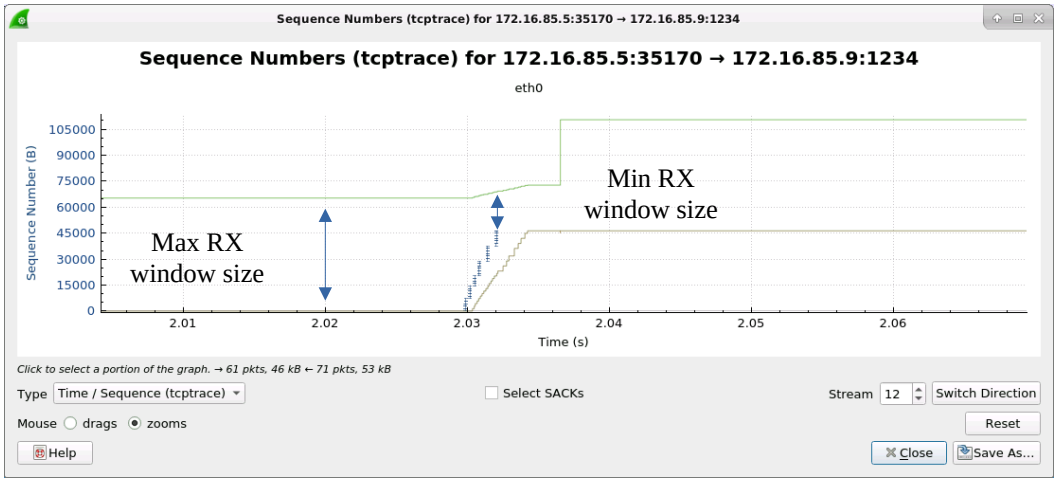


Figure 13 : tcptrace graph

The top green trace in this graph (previously ignored) indicates the servers receive window size i.e. its receive buffer size. The gap between the transmitted data bar and this top trace indicates the spare network capacity. From this graph we can see that the RTT is very small i.e. a packet is transmitted and acknowledged almost immediately i.e. it's difficult to fill the receive buffer, or have a significant number of packets on the wire (in flight), as there is virtually no network delay.

Tip, to only display ACK packets involved in the cowsay transfers for a specific host (IP address will vary) you can use this filter:

```
tcp.port==1234 and ip.src=X.X.X.X and tcp.ack
```

To allow the transmitting host the time to transmit multiple data packets i.e. up to the maximum receive window size, we need to add some latency to the communications link between Pi-1 and Pi-2 using the `tc` command i.e. to have more “in flight” packets. In the Pi-1 SSH window enter the following command:

```
sudo tc qdisc add dev eth0 root netem delay 100ms
```

Re-run the previous task i.e. capture the 600,000 character transfer, and examine the tcptrace graph, as shown in figure 14.

Note, refer to the previous lab for more information on `tc`. To remove this delay enter the command:

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

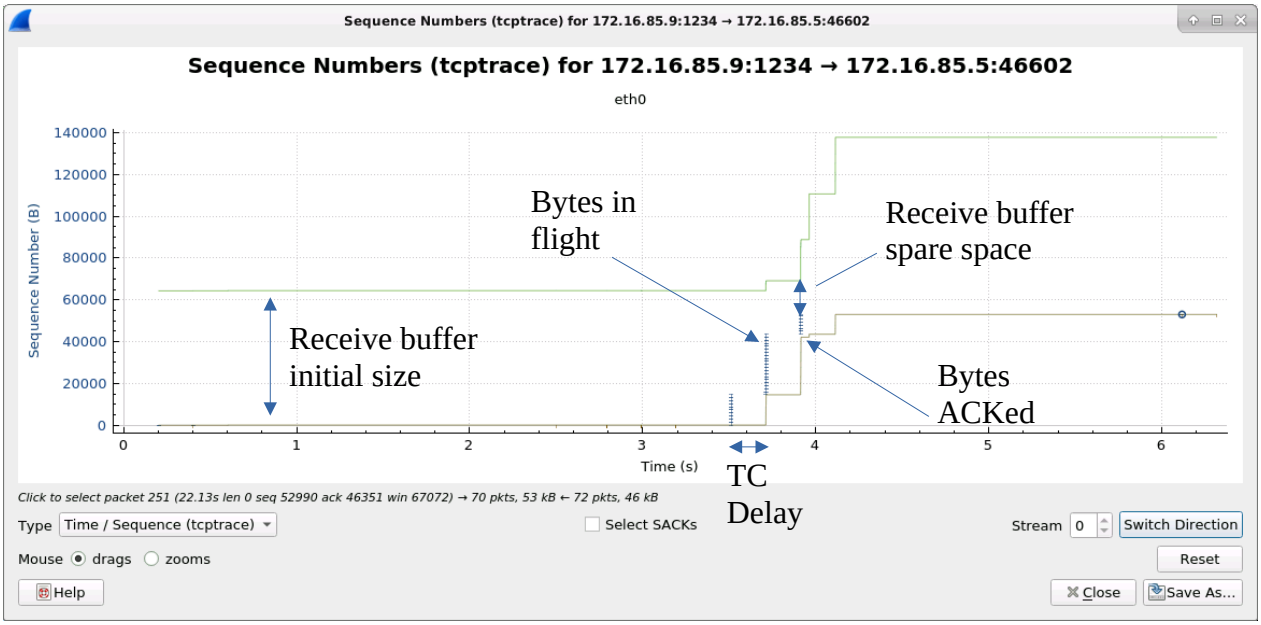


Figure 14 : tcptrace graph for 600,000 characters

From this graph we can see the increase in RTT delay i.e. the visible gap between the blue data packet bars and the grey acknowledged packets trace. We can also see that the server has not been able to fill the clients receive window i.e. we still have spare network capacity to have bytes in-flight.

Note, remember the top green bar is not the receive buffer’s actual capacity on the RX host, it just indicates the last communicated receive window size transmitted from the receiving host in an ACK packet i.e. it is not a real time plot of the receive buffer size :). Also, the the receive buffer’s capacity is the difference between the green and grey traces.

To try and fill network capacity i.e. fill the receive buffer, we need to transmit more data. Therefore,

we either need a longer book, or we can send the book twice. An easy way to do this is to edit the `cowsayBigTest` program as shown below:

```
24      sockTX.send( b"text" )
25      sockTX.send( ''.join(str(x) for x in data).encode('UTF-8')  )
26      sockTX.send( ''.join(str(x) for x in data).encode('UTF-8')  )
27      sockTX.send( b"\n" )
```

Alternatively, we can slow down the rate at which the client reads the data out of the receive buffer. To do this is to edit the `cowsayBigTest` program as shown below i.e. remove the comment:

```
34          time.sleep(0.001)
```

Task : try applying these two different techniques and use Wireshark to observe how they each affect the packets sent onto the network i.e. their resulting tcptrace graphs.

Task 5

The final TCP optimisation technique we will be looking at is congestion control. This is a little tricky to replicate in the lab as it is determined by time-outs and missing ACK packets i.e. we will need to replicate a system that looses packets. To simulate lost packets we can again us `tc`. In the Pi-2 SSH window enter the following command to remove all previous settings :

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

Without packet loss enabled TCP uses its normal “slow start” function i.e. the client sends out exponentially larger blocks of data until flow or congestion control measures kick in. As there are only two hosts on this network, there is no congestion i.e. very difficult to lose ACK packets :). In this setup the main factor that throttles the client’s transmission speed is the servers receive window size.

To set a 100ms latency (delay) and a 10% packet loss rate (probability of dropping a packet) enter the following command in the Pi-2 SSH window:

```
sudo tc qdisc add dev eth0 root netem delay 100ms loss 10%
```

We will also add a delay to Pi-1 to make the trace graph easier to read. Therefore, in the Pi-1 SSH window enter the command below, this will add a 50ms delay to incoming packets on Pi-1:

```
sudo tc qdisc add dev eth0 root netem delay 50ms
```

Task : re-run the previous task i.e. capture 600,000 × 2 character transfers. Select a data packet coming from Pi-1 (client) and examine the tcptrace graph. Select a data packet from Pi-2 (server) and examine its tcptrace graph. Can you spot where errors occur, where packets are lost?

Hints, with 10% packet loss enabled the trace is a lot more interesting. If you examine the new Wireshark packet trace you will see that there are a lot of **BLACK** packets in the lower half of the packet trace i.e. where the server is transmitting back packets to the client. In Wireshark this

indicates an error, consider the examples in figure 16.

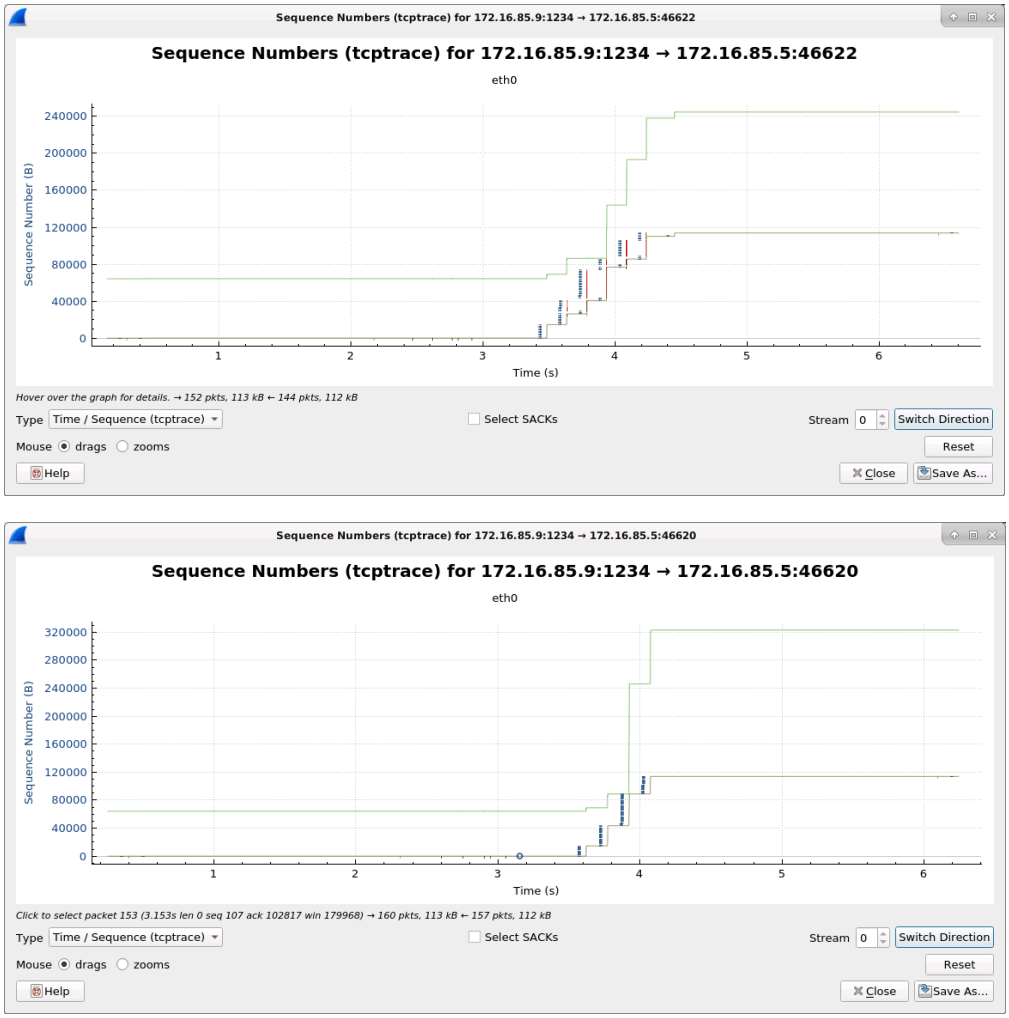


Figure 15 : tcptrace graph, 10% loss (top), 0% loss (bottom)

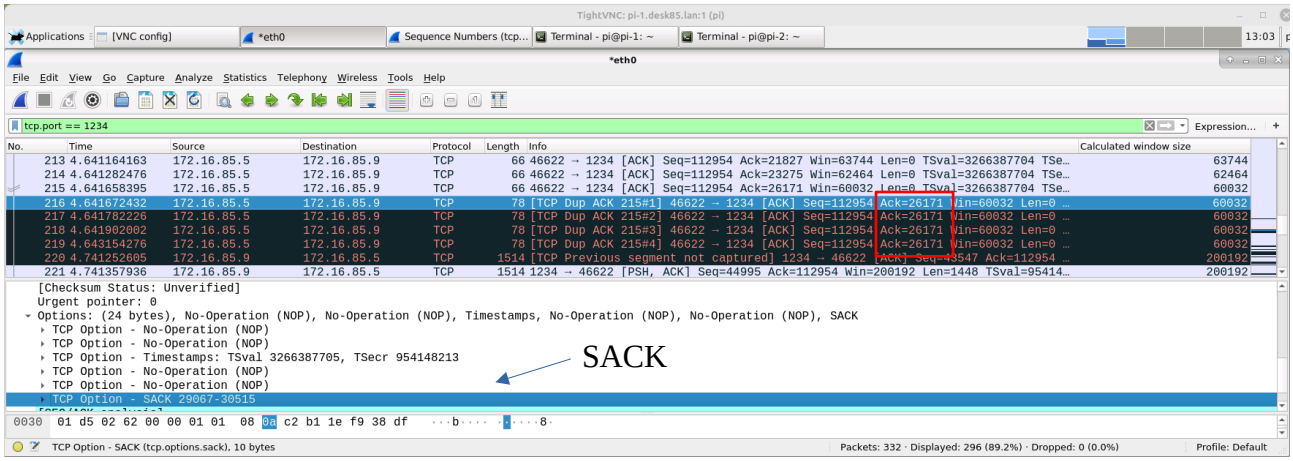


Figure 16 : duplicate ACKs

Note, the packets dropped by Pi-2 will be random, therefore, your packet trace will differ. You may need to re-run the cowsay transfers a couple of time to see all the different scenarios.

In the example shown in figure 16 we loose packet SEQ number 26171+1448=27619. The receiving host signals this to the transmitting host by not updating its ACK number. It will still transmit back ACK packets for each data packets it receives, but the ACK number is fixed to 26171 i.e. the last element in the stream successfully received.

A key point to remember is that the transmitting host will only start to know that an error has occurs when it receives back a duplicate ACK packet i.e. multiple ACK packets with the same number. This information will be delayed by the RTT, so the transmitting host will keep transmitting packets for some time, so you can get a lot of duplicate ACKs. The receiving host will ACK each new data packet it receives with the same ACK number, until the transmitting host gets the message and retransmits the missing packet as shown in figure 17.

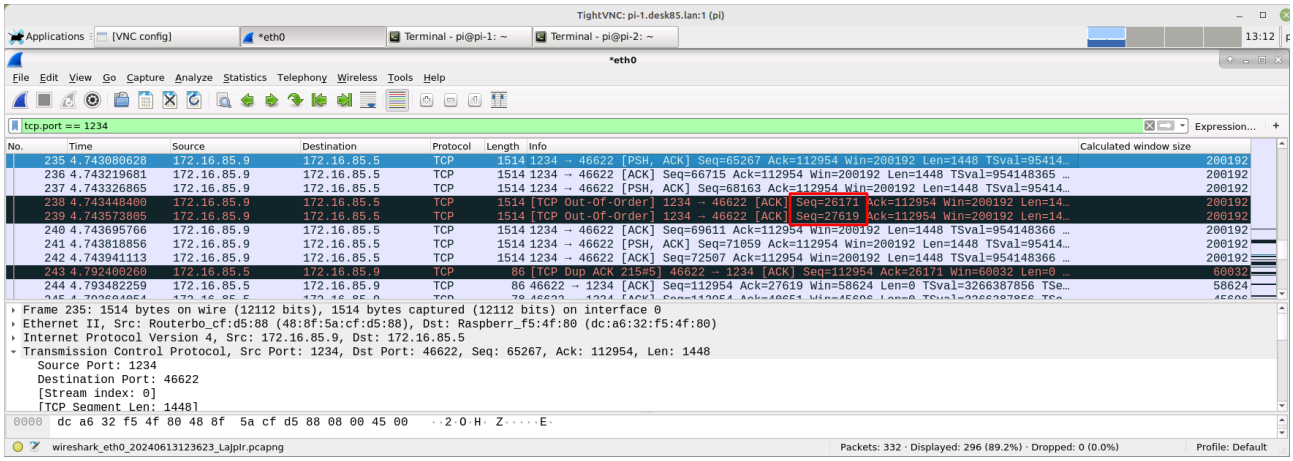


Figure 17 : retransmission

Reading through and analysing packet traces is confusing i.e. identifying the interactions between client and sever, what has caused the “error”and when it is corrected. It can be useful to switch between this packet trace view and the tcptrace graph e.g. the example in figure 18 to help identify what is happening on a network. The sequence of packets shown in figures 16 and 17 can be broken down into the following phases:

1. The server puts a block of data onto the network, this is successfully received and acknowledged.
2. As no error occurred the server doubles the size of the data block, this is transmitted, however, two packets are lost. At this point the client sends duplicate ACKs i.e. select ACKs (SACK) acknowledging the data received after this loss has been successfully received.
3. The server receives a DUP ACK, identifies what SEQ numbers have been lost and then retransmits this data.

Note, the grey ACK trace in the tcptrace graph does not step up until the missing packets are retransmitted. To further improve network efficiency most systems will use a technique based on a sliding window i.e. again to try and minimise the “stop” time. This is not covered in this module,

but, a nice video to watch later that discusses this concept is available here:

https://www.youtube.com/watch?v=klDhO9N01c4&t=89s

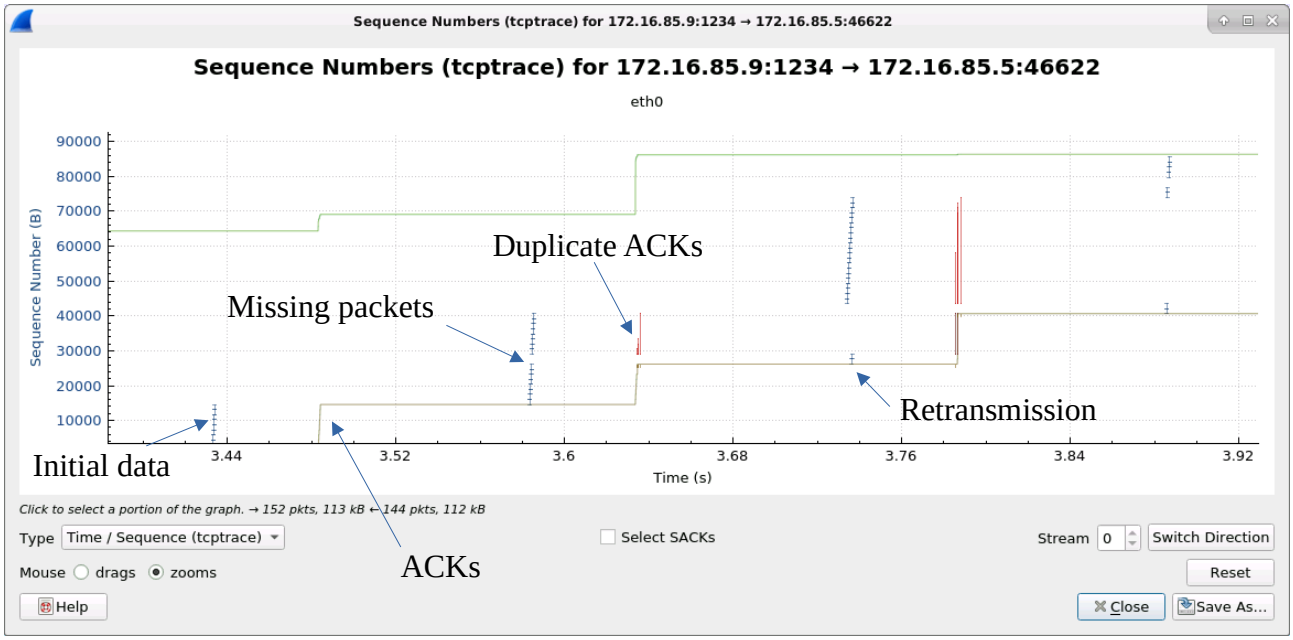


Figure 18 : tcptrace graph for figures 16 and 17

Task : using the lossy network, examine the packet capture in Wireshark, can you identify a duplicate ACK packet i.e. when a packet is lost packet on the network?