SYS2 (NET) Laboratory 7 : IP and Routing

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 look in more detail at the protocol we have been using from the start of this module i.e. the Internet Protocol (IP). Before starting this lab make sure you have watched video **Lecture 4A part 1 & 2**. We are finally moving down the protocol stack shown in figure 1 into layer 3, the Network layer. At this level we are no longer looking at how applications or hosts communicate, rather the "simpler" problem of how do we get (route) our network packets from one IP address to another e.g. how did the ping packets from lab 1 find their way from York to Italy? As you probably guessed routing packets between the network of networks that is the Internet is a little complex, here be dragons :). Therefore, for the purposes of this module we are going to limit our investigation into routing to networks like the one we are using in the computer science building i.e. thousands of network connections rather than millions. In this lab we will first examine the IP protocol, its purpose and how it is used in conjunction with TCP and UDP. Next we will consider how IP packets are routed through a network e.g. gateways and manual routing rules. In the next laboratory we will also be looking at routing protocols that automatically update a router's routing rules. At the end of this practical you will understand how :

- large packets are fragmented i.e. broken down into smaller blocks and reassembled.
- to identify the default route into and out of a network i.e. its gateway.
- IP packets are routed across a network and how static routing rules can be defined.
- to access and configure the mikroTik router.

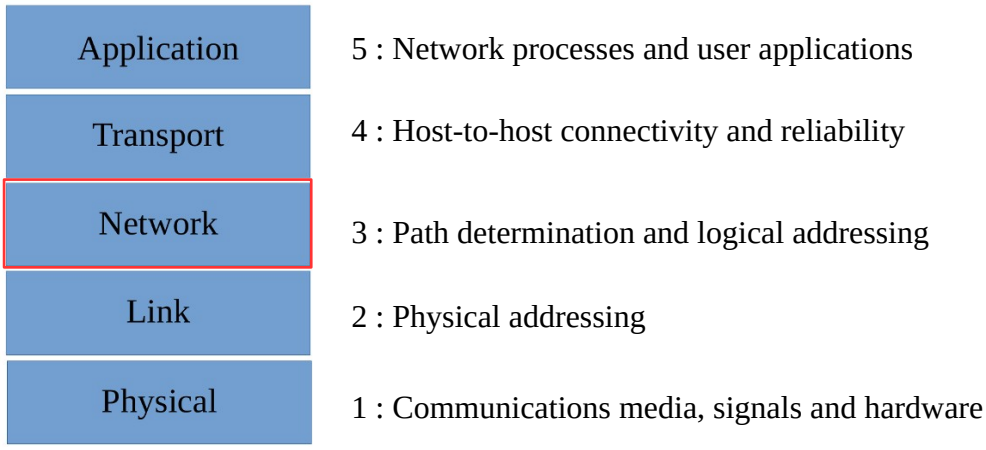


Figure 1 : The Internet protocol stack

Task 1

The Internet protocol's packet is called a datagram, as shown in figure 2. This protocol has been around since the start of the Internet, so some of the header fields defined have more significance than others i.e. that's why we have IPv6 :). The main functions implemented by this protocol are logical addressing (identifying hosts), and fragmentation (transporting data across a network).

For a complete description of each block/field refer to RFC 791:

https://datatracker.ietf.org/doc/html/rfc791

Task : do have a quick read through this RFC to get a feel about how the IP protocol is used.

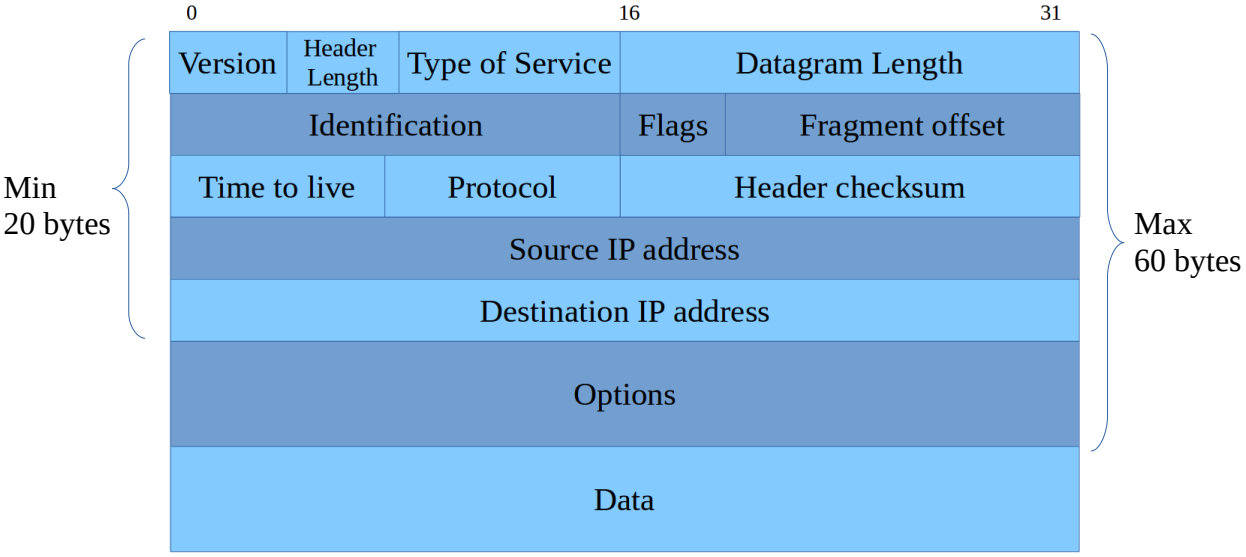


Figure 2 : internet protocol header

A core function of the IP protocol is fragmentation. This occurs when higher layer protocols require a segment size larger than the network can support. In this situations the IP protocol automatically breaks these larger packets down into smaller fragments that can be transmitted across the network. These fragments are then reassembled by the receiving host.

When you look at the IP header block diagram in figure 2, you can see that the datagram length field i.e. header + data, is represented using 16bits. Therefore, in theory the IP protocol can use datagrams up to 65535 bytes. However, these “jumbo” datagrams require specialised network hardware (routers) that are capable of storing and forwarding these large blocks of data.

Note, the original IP specification assumed 576 byte datagrams and that this size should only be increased if the destination host is prepared to accept these larger datagrams (MSS and MTU).

The advantage of using a larger datagram sizes is efficiency i.e. the ratio between header and data. The larger the data compared to headers, the higher the efficiency, the higher the effective bandwidth, as shown in figure 3.

MTU	IP header	TCP header	Data	Efficiency	Type
576	20	20	536	93.1%	Min
1500	20	20	1460	97.3%	Normal
9000	20	20	8960	99.5%	Jumbo
64000	20	20	63960	99.9%	Max

Figure 3 : datagram efficiency

To show this in action consider the software used to transfer the image of Bob, shown in figure 4. This program uses the UDP protocol, the segment being fixed by the constant BUF_SIZE.

```
import socket
import os
import time

UDP_IP = "192.168.X.254"
UDP_PORT = 8000
BUF_SIZE = 1024

sockTX = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

read_file_name="bob.jpg"
write_file_name="bob-copy.jpg".encode("ascii")
file_size = os.path.getsize(read_file_name)
number_of_segments = str(-(-file_size//BUF_SIZE)).encode("ascii")

sockTX.sendto(write_file_name, (UDP_IP, UDP_PORT))
time.sleep(0.01)
sockTX.sendto(number_of_segments, (UDP_IP, UDP_PORT))
time.sleep(0.01)
print("Send: " + read_file_name + ":" + str(number_of_segments))

f=open(read_file_name,"rb")
data = f.read(BUF_SIZE)

for i in range(0,number_of_segments):
  sockTX.sendto(data, (UDP_IP, UDP_PORT))
  time.sleep(0.01)
  print("tx segment: " + str(i) + " len: " + str(len(data)))
  data = f.read(BUF_SIZE)

sockTX.close()
f.close()
```

Figure 4 : UDP fragmentation code

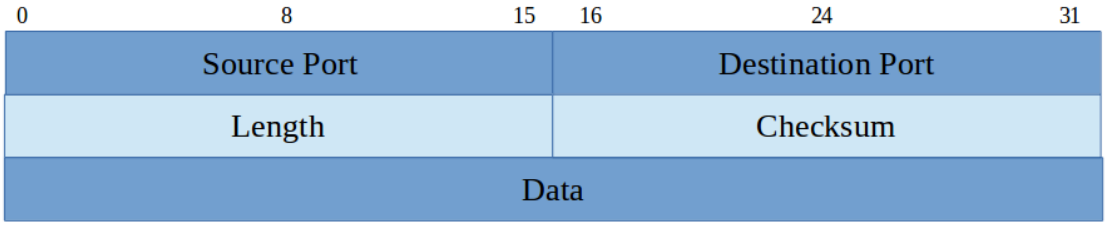


Figure 5 : UDP segment

The current buffer size is set to 1024 bytes i.e. to fit within a typical IP datagram. However, as shown in figure 5 the UDP segment length is also represented by a 16bit value. Therefore, in theory

we can transmit the image of Bob in a single network “packet”, as the image of Bob is approximately 21KB. To see fragmentation in action we will re-use the code from laboratory 4 and transfer an image of Bob from Pi-1 to Pi-2, as shown in 6.

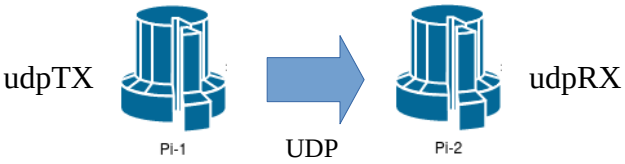


Figure 6 : image transfer route

Download the python programs `udpRX.py` and `udpTX.py` from the VLE . Update these programs to reflect the IP addresses / ports used (as always X=Box/Desk) :

- `udpTX.py: UDP_IP = "192.168.X.2" UDP_PORT = 8000`
- `udpRX.py: UDP_IP = "192.168.X.2" UDP_PORT = 8000`

Using FileZila transfer these programs to Pi-1 and Pi-2. Next, on the PC click on the start button and select the command prompt.

`-> Command Prompt`

then SSH into Pi-1 by entering the command below:

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

Then open a second command prompt on the PC and SSH into Pi-2 by entering the command below:

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

In the Pi-2 SSH terminal run the command:

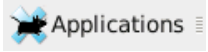
```
python3 udpRX.py
```

Then in the Pi-1 SSH terminal run the command:

```
python3 udpTX.py
```

If all goes correctly this will transfer the image of Bob across the network, from Pi-1 to Pi-2, writing the data to the file `bob-copy.jpg`. Using FileZila download this file to the PC to confirm that it was transferred correctly.

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 1 (`eth1`) in the Interface list, next enter the

packet protocol filter :

```
ip.addr==192.168.X.1 and !tcp and !ntp          (X=Box/Desk)
```

and click on the Apply button (this will remove most of the network “noise”). Next click on the Start icon on the top toolbar in Wireshark. Restart the RX and TX programs to re-transfer the image of Bob.

You will now see some activity in the main window as Wireshark captures the packet associated with this data transfer. Finally, click on the Stop icon on the top toolbar to stop capturing network traffic. What you should observe in the Wireshark packet trace is that as expected the image of Bob is transferred using multiple 1024B UDP segments that fit within a single MTU sized IP packet.

Task : increase the constant `BUF_SIZE` to 9000 in the RX and TX programs. Then repeat this transfer from Pi-1 to Pi-2. How many UDP data packets are used to transfer the image? Can you see how the UDP packet is broken down into IP packets?

What you should now observe is that each UDP packet is broken down into a series of fragmented IP packets as shown in figure 7. To allow these fragments to be reassembled back into the original UDP packet each of these fragments has the same 16bit Identification code (field shown in figure 2). Also contained within the IP header is the fragment offset i.e. a byte counter, so that the receiving host can re-assembly these fragments correctly even if they are received out of order .

Note, a typical sequence for the fragment offset will be : 0, 1480, 2960, 4440 …

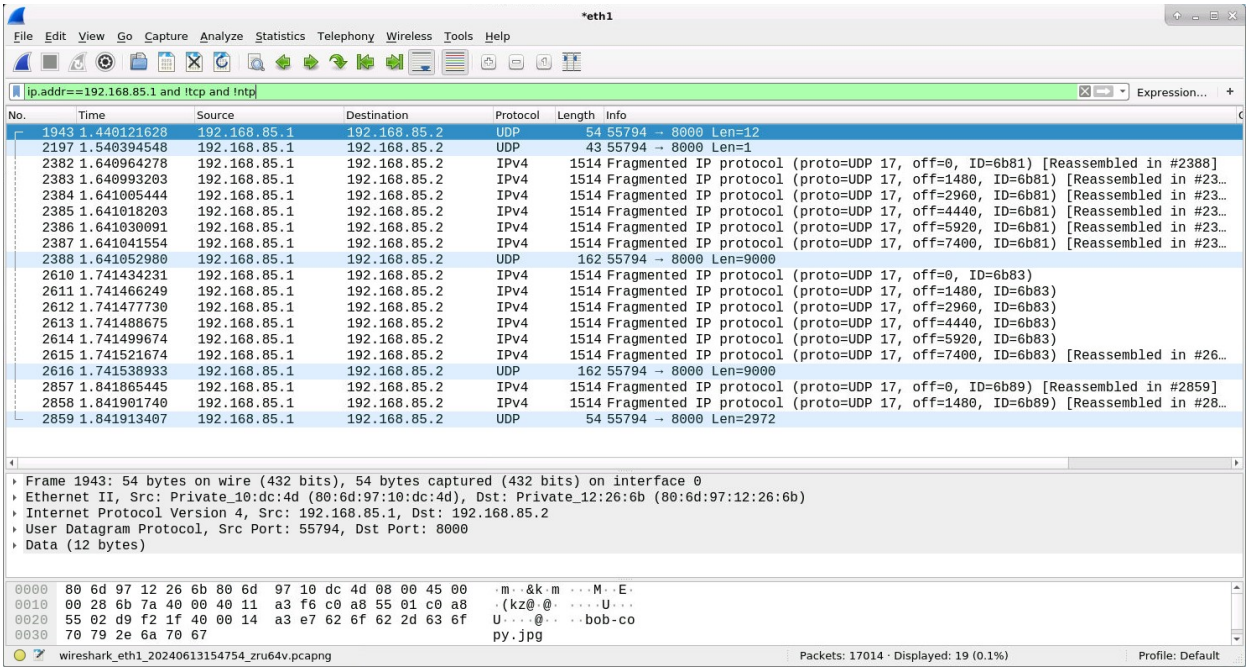


Figure 7 : IP fragments

Task 2

To understand how an IP datagram is routed across a network we need to look at the routing tables on each Pi and the routers they are connected to. Therefore, in the Pi SSH terminals enter the following command :

```
route -n
```

This tables lists the network addresses this Pi knows about, as shown in figure 8. This information is then used to route a packet to the correct physical interface e.g. `eth0`, `eth1` etc.

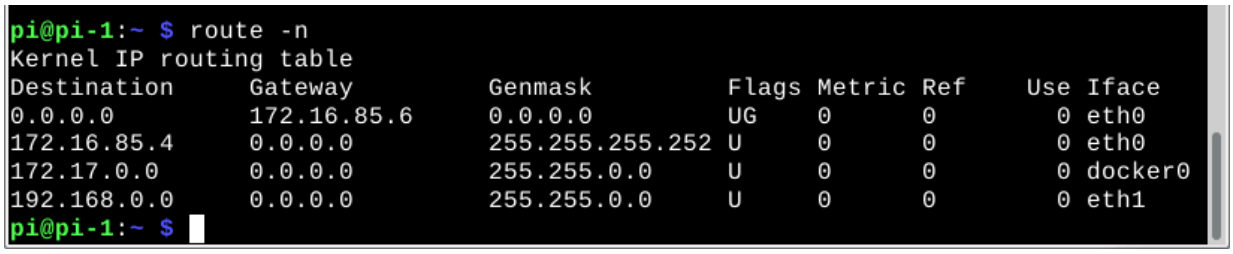


Figure 8 : routing table

Its not just hosts that have IP addresses, networks also have IP addresses i.e. the Destination column in figure 8. These are calculated using a network mask (Genmask or Netmask), typically these end with a `.0`, but this is not always true as you can see for `eth0`, it all depends on what bits within the 32bit IP address identify the network and what bits identify the host.
When reading this table you first need to understand the “special” addresses / values used :

- Address 0.0.0.0 i.e. all 0’s, can be used to signal an invalid or unknown address. A good example is in an initial DHCP handshake when a host does not know what its IP address is.
- When specified as a source address 0.0.0.0 means all valid interfaces e.g. to allow the cowSay server to listen on all network interfaces the following commands are used:

  ```
  sockRX = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
  sockRX.bind((0.0.0.0,1234))
  ```

- Address 255.255.255.255 i.e. all 1s, is the broadcast address i.e. all hosts receiving a packet with this destination address will process this packet. Again, we saw this address in action when we looked at DHCP. We will look at broadcast addresses in more detail in the next lab.
- A netmask of 0.0.0.0 i.e. /0, will match any host address. Think of this 32 bit value as a bitwise AND mask specifying what “network” bits will be used when comparing the destination address contained within the network packet and the network address in the routing table, i.e. what bits have to match.
- A netmask of 255.255.255.255 i.e. /32 network, will match only one host address.

When reading a routing table you are looking for the best match to determine what network interface will be used to forward a packet i.e. typically you will have multiple matches, but its the longest match that wins. Consider the routing table in figure 8. If an application wants to send a packet to address 192.168.100.3 it will apply the listed network masks (Genmasks) to identify the destination network, as shown in figure 9.

Default (GW)

Host address : 192.168.100.3	11000000 10101000 01100100 00000011	
Network mask : 0.0.0.0	00000000 00000000 00000000 00000000	
Result : 0.0.0.0	00000000 00000000 00000000 00000000	
Network : 0.0.0.0		✓

Docker0

Host address : 192.168.100.3	11000000 10101000 01100100 00000011	
Network mask : 255.255.0.0	11111111 11111111 00000000 00000000	
Result : 192.168.0.0	11000000 10101000 00000000 00000000	
Network : 172.17.0.0		✗

ETH1

Host address : 192.168.100.3	11000000 10101000 01100100 00000011	
Network mask : 255.255.0.0	11111111 11111111 00000000 00000000	
Result : 192.168.0.0	11000000 10101000 00000000 00000000	
Network : 192.168.0.0		✓

ETH0

Host address : 192.168.100.3	11000000 10101000 01100100 00000011	
Network mask : 255.255.255.252	11111111 11111111 11111111 11111100	
Result : 192.168.100.0	11000000 10101000 01100100 00000000	
Network : 172.16.85.4		✗

Figure 9 : network calculation

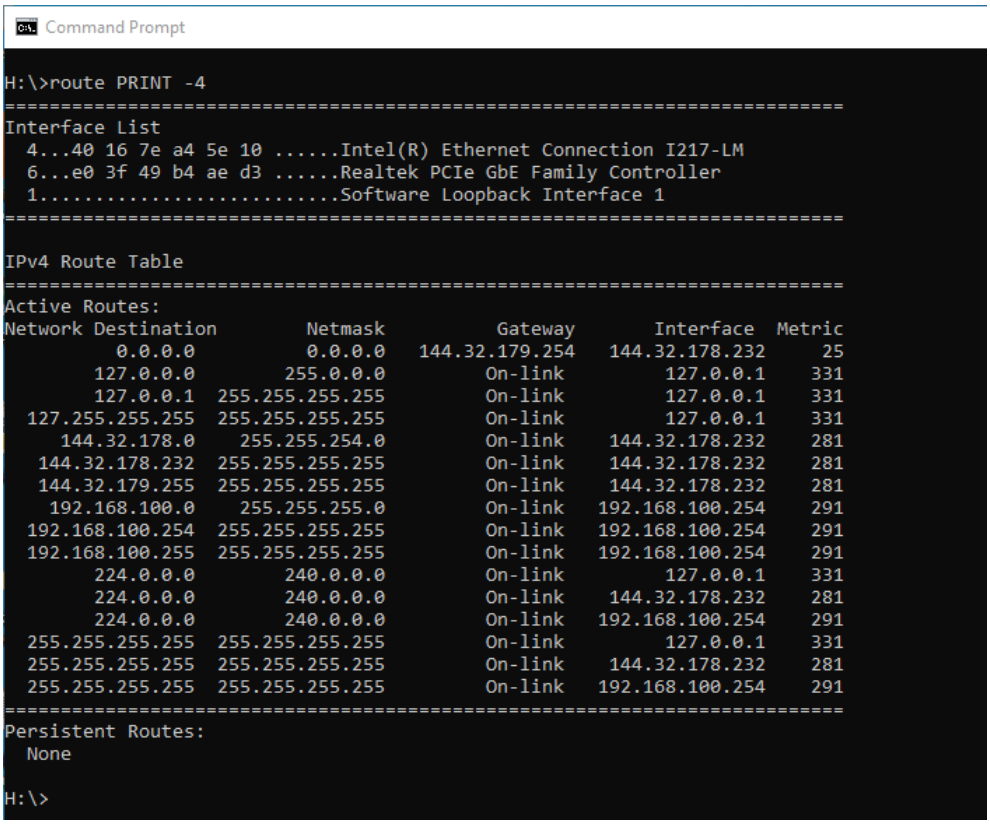


Figure 10 : PC routing table

The results from these bitwise AND calculations are then compared to the network addresses listed in the destination column. In this example the longest match is `eth1`. Therefore, the packet is transmitted on this interface. However, you can also see it matches to destination address 0.0.0.0 i.e. the shortest match. However, this is ignored as we have a longer matching network address.

Note, most networks have a default route i.e. a gateway (GW), a network were packets are sent when the destination address does not match any known local networks. In this example the default gateway is the router (mikroTik) connected to `eth0` and is indicated in the routing table by the `G` flag.

You can also see the routing table for the PC. On the PC click on the start button and select the command prompt.

```
-> Command Prompt
```

Then enter the command :

```
route PRINT -4
```

This will display the network routing table for your PC, an example is shown in figure 10.

Note, the metric column is an indication of “cost”, this is used to select between networks when you have multiple valid (equal length) routes to a given destination. Calculation of this “cost” can be based on the number of “hops” i.e. routers between you and the destination, or associated latency, throughput or reliability. The higher the metric number the higher the “cost”. Therefore, in the case of multiple valid networks the lowest metric value, the lowest cost network would be selected.

Task : examine the routing table on Pi perform the network mask calculation on the IP address : `10.0.0.1`, to determine what interface this packet will be forwarded to.

Task 3

The concept of addressing i.e. how we identify hosts on a network, highlights a key concept used in networking protocols, that of encapsulation. Up to this point we have only considered how data is transferred between hosts at the transport layer i.e. port numbers. However, in the network layer we now need to consider how we get the information contained within the UDP/TCP headers from one host to another i.e. how we encapsulate different protocols within others, as shown in figure 11.

The key idea behind a protocol stack is to hide detail i.e. at each layer we only consider the information needed to perform the tasks associated with that layer. When data is transmitted, it goes down the protocol stack. The previous layer’s packet being encapsulated within a new packet containing the information needed to allow it to continue it’s journey across the network. When data is received data goes up the protocol stack. The previous layer’s header being removed until the final destination is reached i.e. an application on a host.

Note, ideally these layers will be self contained i.e. no leakage of information between layers. This allows different protocols to be replaced as needed. Consider the link layer in figure 11. In this example we are using Ethernet frames, a wired (copper) connection. If we want to go to a wireless

connection all we need to do is replace this layer with the appropriate wireless communications protocol i.e. the Network (IP) and Transport (UDP/TCP) layers are unaffected. Although desirable this isolation is not always practical and leakage between layers can occur e.g. UDP checksum.

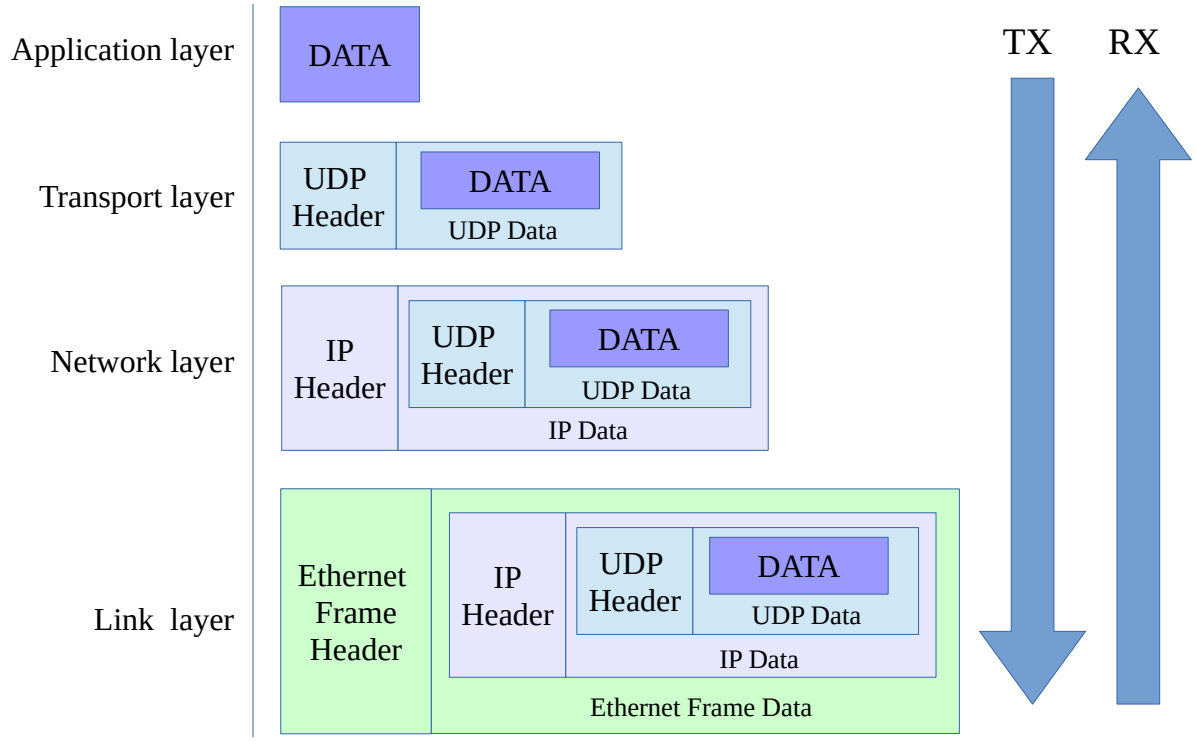


Figure 11 : encapsulation

To illustrate encapsulation and routing in action we can configure a series of tunnels between the PC and the Pi system. This is comparable to the system we created in laboratory 4, but rather than using `nc` and `ssh`. To create these communication paths we can use point-to-point tunnels (`tun0` and `tun1`) and static routing rules to route these packets along this chain, as shown in figure 12.

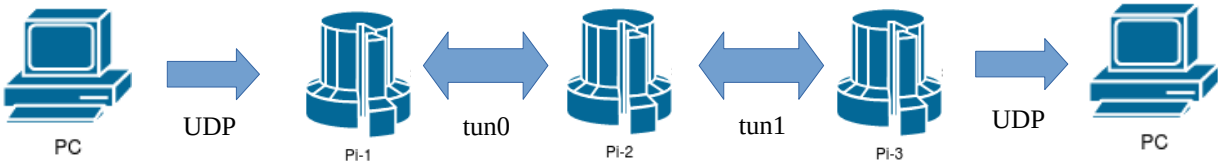


Figure 12 : image transfer route

We will again transmit an image of Bob from the PC, across the Pi network and then display the received image on the PC, but this time using network routing rules. On the PC click on the start button and select the command prompt.

```
-> Command Prompt
```

then SSH into Pi-1 by entering the command below:

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

Then open a second command prompt on the PC and SSH into Pi-2 by entering the command below:

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

Before adding these tunnels we can first check the existing network interfaces. In the Pi-1 SSH terminal enter the following command:

```
ifconfig
```

This will return the active network interfaces e.g. our normal network interfaces `eth0`, `eth1` and `lo`. The interface `lo:1` is a secondary IP address that is assigned to the loop back interface `lo` (internal network) i.e. an alias, as shown in figure 13.

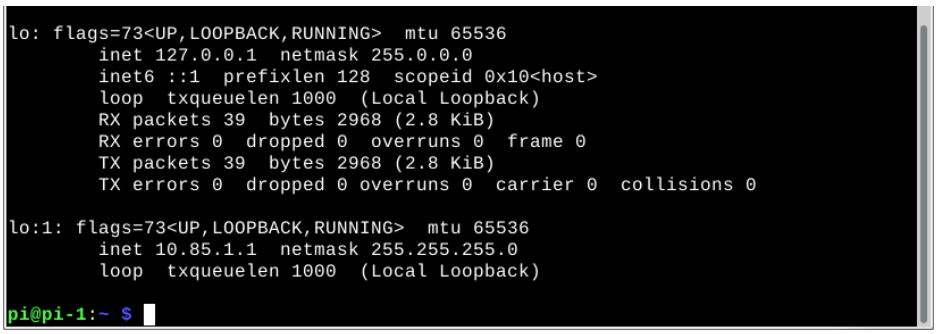


Figure 13 : loop back interface

Note, you can assign multiple IP addresses to any NIC i.e. create multiple virtual network interfaces for each physical network interface. Why? It's sometimes useful to simplify other tasks e.g. allow different network access to different users or applications.

In addition to creating virtual interfaces the OS also allows us to create virtual tunnels and bridges. A tunnel allows a user to define a new virtual network interface. This can be used to link Pi-1 to Pi-2 using the existing physical network interfaces. However, the IP addresses sent across this tunnel can be a different from the physical IP addresses used to transport them i.e. a virtual connection or tunnel. Confusing I know, but stick with it :).

Task : to create a new tunnel network interface, enter the following commands in the Pi-1 SSH terminal :

```
sudo ip tunnel add tun0 mode ipip local 172.16.X.5 remote 172.16.X.9
sudo ip link set tun0 up
sudo ip addr add 10.X.1.2/24 dev tun0
```

Note, as always the IP address of your system will vary, X=Box/Desk.

The tunnel uses IP-over-IP to transfer the UDP packets that transfer the image of Bob i.e. we encapsulate an IP datagram, within an IP datagram. The outer IP datagram contain the remote (destination) and local (source) IP addresses of Pi-2 and Pi-1. The inner IP datagram uses the network address assigned to the virtual loop back interface lo:1, as shown in figure 14.

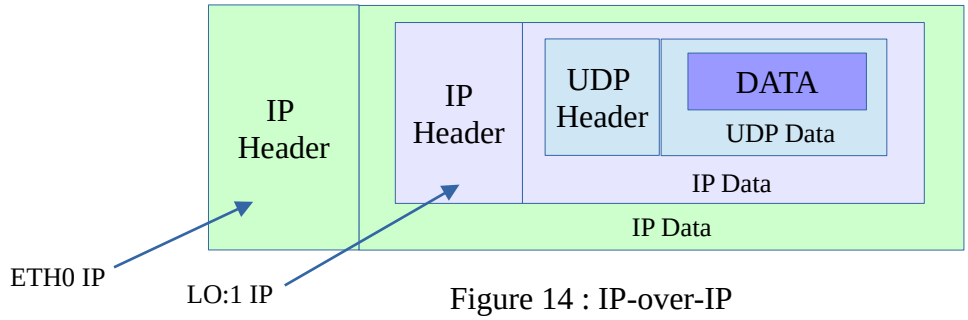


Figure 14 : IP-over-IP

Note, the layer 3 IP datagram shown in figure 14 will again be wrapped in the layer 2 protocol used by the network e.g. if its copper then this would typically be the Ethernet protocol, we will be looking at this protocol in a later lab.

Task : on Pi-2 and create a new tunnel network interface and enter the following commands in the Pi-2 SSH terminal :

```
sudo ip tunnel add tun0 mode ipip local 172.16.X.9 remote 172.16.X.5
sudo ip link set tun0 up
sudo ip addr add 10.X.2.2/24 dev tun0
```

To test that these new tunnel interfaces have been configured correctly run `ifconfig` on Pi-1 and Pi-2. If all is good you should see these new interfaces. An example is shown in figure 15.

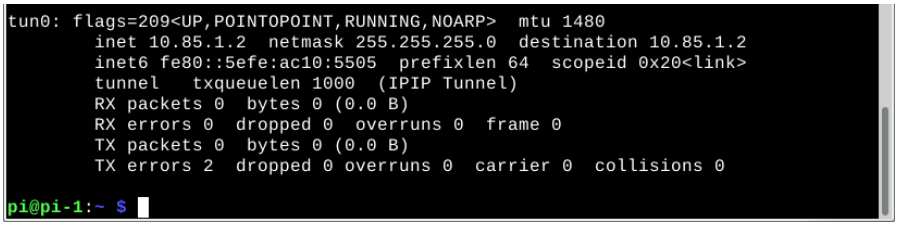


Figure 15 : tunnel interface Pi-1

Task : to test if this tunnel is working correctly we can ping the network interfaces used by these tunnels. In the Pi-1 SSH terminal ping the IP addresses below, then repeat these tests in the Pi-2 SSH terminal.

- 172.16.X.5 : Pi-1 eth0
- 172.16.X.6 : Router port 1
- 172.16.X.9 : Pi-1 eth0
- 172.16.X.10 : Router port 2
- 10.X.1.1 : Pi-1 lo:1
- 10.X.1.2 : Pi-1 tunnel interface
- 10.X.2.2 : Pi-2 tunnel interface
- 10.X.2.1 : Pi-2 lo:1

Note, some of these Pings will **NOT** work

As you would expect each Pi can “see” its interfaces, but when you ping the remote lo:1, the remote tunnel interface you receive the message “Destination Net Unreachable”, as shown in figure 16.

```
pi@pi-1:~ $ ping 10.85.2.2
PING 10.85.2.2 (10.85.2.2) 56(84) bytes of data.
From 172.16.85.6 icmp_seq=1 Destination Net Unreachable
From 172.16.85.6 icmp_seq=2 Destination Net Unreachable
From 172.16.85.6 icmp_seq=3 Destination Net Unreachable
^C
--- 10.85.2.2 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 93ms

pi@pi-1:~ $
```

Figure 16 : unreachable network

This is due to how routing is performed on each Pi i.e. when the `ping` application uses each IP address the OS will first look at its routing table to see what network interface it should pass this packet to.

Task : examine the updated routing tables on Pi-1 ans Pi-2 using the `route` command and identify the tunnel network addresses. Knowing how routing calculations are performed consider how the OS will route packets between Pi-1 to Pi-2 i.e. through our tunnel interfaces. Where will the following IP addresses be sent?

- 10.X.1.1 : Pi-1 lo:1 (X=Desk/Box)
- 10.X.1.2 : Pi-1 tunnel interface
- 10.X.2.2 : Pi-2 tunnel interface
- 10.X.2.1 : Pi-2 lo:1

What your calculations should show is that when we try to ping 10.X.2.2 or 10.X.2.1 these network addresses will be matched to the default gateway (GW) and not our tunnel. Therefore, on Pi-1 we need to add a routing rule to tell the Pi to send these packets to the tunnel rather than the gateway.

Task : in the Pi-1 SSH terminal enter the following command:

```
sudo ip route add 10.X.2.0/24 dev tun0
```

Note, as always the IP address of your system will vary, X=Box/Desk.

To see how this command has updated the routing table, we again enter the command :

```
route -n
```

You should now see a new rule, as shown in figure 17.

```
pi@pi-1:~ $ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.16.85.6     0.0.0.0         UG    0      0        0 eth0
10.85.1.0       0.0.0.0         255.255.255.0   U     0      0        0 tun0
10.85.2.0       0.0.0.0         255.255.255.0   U     0      0        0 tun0
172.16.85.4     0.0.0.0         255.255.255.252 U     0      0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
192.168.0.0     0.0.0.0         255.255.0.0     U     0      0        0 eth1
pi@pi-1:~ $
```

Figure 17 : updated routing table

Task : to test this tunnel, in the Pi-1 SSH terminal ping the IP addresses 10.X.2.2 or 10.X.2.1 on Pi-2. Can you now reach these networks? Repeat this process on Pi-2 i.e. can you also ping the IP addresses 10.X.1.2 or 10.X.1.1 on Pi-1?

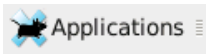
What you should observe is that Pi-1 can still not ping Pi-2, but now the Pi-1 “destination network unreachable” error message is not displayed, rather the ping command “freezes”, indicating that it has passed the packet forwards, but it has not received a response i.e. Pi-2 receives the ping request but does not know where to send the response.

Task : to fix this problem, in the Pi-2 SSH terminal add a routing rule to allow Pi-2 to ping the IP addresses 10.X.1.2 or 10.X.1.1 on Pi-1. Enter this routing rule and test if it functions correctly. If needed you can remove any incorrect routing rules you would use the `del` command as shown in the example below:

```
sudo ip route del 10.X.Y.0/24
```

Note, if you would like to check your answer a solution is shown in Appendix A.

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, then select the network interface Ethernet 0 (`eth0`) in the Interface list. In the Wireshark GUI enter the packet protocol filter for IP :

```
ip and !dns
```

and click on the Apply button. To send some test packets through this tunnel we can use `nc` command discussed in laboratory 4. In the Pi-2 SSH terminal enter the following command:

```
nc -u -l -p 8080
```

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

```
echo "hello" | pv | nc -u 10.X.2.1 8080        (where X=Box/Desk)
```

You will now see some activity in the main window as Wireshark captures the packet associated with this data transfer, as shown in figure 18. Finally, click on the Stop icon  on the top toolbar to stop capturing network traffic.

Note, you should also see the message “hello” displayed in the Pi-2 SSH terminal. Press `CTRL C` to exit `nc`.

Task : look through the protocol stack captured in Wireshark, as shown in figure 18. Can you see how the network packet is constructed i.e. its layers? Can you see that we have two sets of Source and Destination IP addresses? If needed have a look back at figure 11 to refresh your memory

regarding the different layers. The key section is highlighted in **RED** i.e. we now have two IP layers i.e. we are carrying IP packets over IP, we have encapsulated an IP within an IP packet :).

Note, point-to-point tunnels like this are a useful way to transport packets between two parts of a network that don't have a direct connection e.g. a company network spread across two building in different parts of the country. We can now transport these packets across the Internet, but from the hosts point of view they look like they are on the same network.

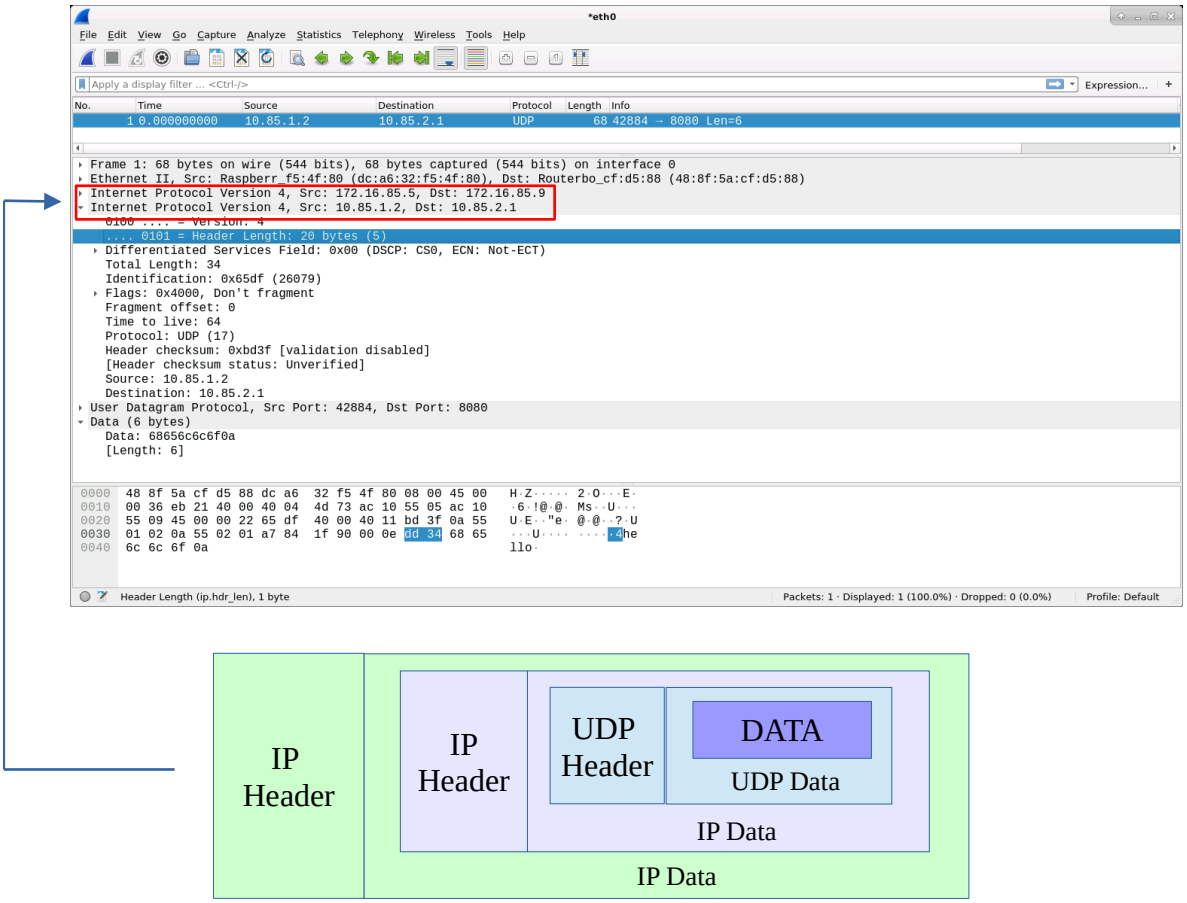


Figure 18 : network encapsulation, IP-over-IP

Task 4

Using `tun0` we have created a point-to-point connection between Pi-1 and Pi-2. We can see how IP packets are routed through this network using the destination IP address and the routing table rules stored in each host or router. Lets now expand this network by adding a second tunnel `tun1` from Pi-2 to Pi-3, as shown in figure 19.

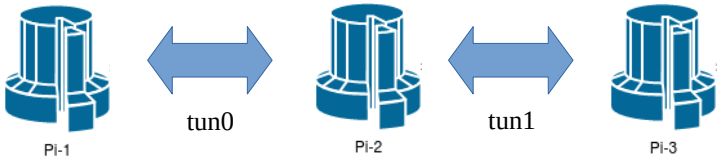


Figure 19 : image transfer route

Task : to add this second tunnel `tun1` we could repeat the same process previously described in Task 3 i.e. add a new `tun1` interface on Pi-2 with the local IP address 10.X.2.2 and a remote IP address 10.X.3.2. However, can you spot the "issue" in doing this?

Hint, consider the routing table on Pi-2, what will be the routing rules for subnet 10.X.2.0/24?

The issue is that this will create two identical routes for 10.X.2.0/24, each pointing to a different interface: `tun0` and `tun1`, as shown in figure 20.

```
pi@pi-2:~ $ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.16.101.10   0.0.0.0         UG    0      0        0 eth0
10.101.1.0      0.0.0.0         255.255.255.0   U     0      0        0 tun0
10.101.2.0      0.0.0.0         255.255.255.0   U     0      0        0 tun0
10.101.2.0      0.0.0.0         255.255.255.0   U     0      0        0 tun1
10.101.3.0      0.0.0.0         255.255.255.0   U     0      0        0 tun1
172.16.101.8    0.0.0.0         255.255.255.252 U     0      0        0 eth0
192.168.0.0     0.0.0.0         255.255.0.0     U     0      0        0 eth1
pi@pi-2:~ $
```

Figure 20 : routing table ambiguity for desk 101 :(

The OS will treat both interfaces as valid paths to the same subnet, resulting in duplicate routes. We can add metrics or routing policies to overcome come this, but its best to avoid these types of situations :). Therefore, the simplest solution is to place these tunnels on different networks. Consider the set of commands below used to create the `tun1` interface on Pi-2:

```
sudo ip tunnel add tun1 mode ipip local 172.16.X.9 remote 172.16.X.13
sudo ip link set tun1 up
sudo ip addr add 10.X.3.3/24 dev tun1
```

Task: if these commands are used how will Pi-2's routing table be updated? Does this remove the previous problem? Do we still need to add explicit routing rules to Pi-2 to pass traffic going to the 10.X.3.0/24 network?

Task : using this knowledge implement the tunnel `tun1` from pi-2 to pi-3 **AND** update the routing tables for pi-1, pi-2 and pi-3, such that Pi-1 can ping the network interface 10.X.3.1 on Pi-3.

Hint, if you would like to check your answer one possible solution is described in Appendixes B, C and D. To help avoid typos and to speed up the entry of these commands, this process can be implemented using scripts, for more information on how to do this refer to Appendix E.

Task : to test if your tunnels are working, perform the following tests:

- Can Pi-1 ping Pi-2's internal IP address: 10.X.2.1?
- Can Pi-2 ping Pi-1's internal IP address: 10.X.1.1?
- Can Pi-2 ping Pi-3's internal IP address: 10.X.3.1?
- Can Pi-3 ping Pi-2's internal IP address: 10.X.2.1?
- Can Pi-1 ping Pi-3's internal IP address: 10.X.3.1?

- Can Pi-3 ping Pi-1's internal IP address: 10.X.1.1?

If the answer to each of these questions is Yes, your system is working correctly :).

We can now connect the PC to the head (Pi-1) and tail (Pi-3) of this image transfer route. Using `SSH` terminals on Pi-1 and Pi-3 run these commands :

```
Pi-1 :   nc -u -l -k -p 8000 | pv | nc -u 10.X.3.1 8000
Pi-3 :   nc -u -l -k -p 8000 | pv | nc -u 192.168.X.254 8000
```

Then using the `udpTX` and `udpRX` programs transmit the image of Bob across this series of tunnels.

IMPORTANT, remember to set the python code packet size back to 1024.

Task : can you transfer the image of Bob across this network i.e. from the PC across the two tunnels and back to the PC?

WebFig tab

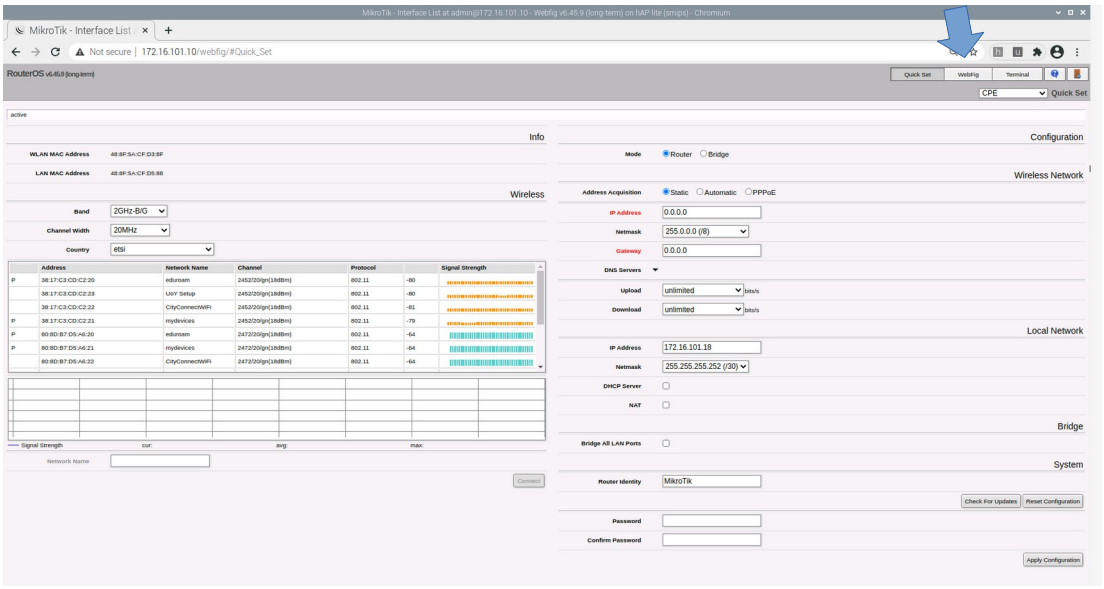


Figure 21 : mikroTik web interface

Task 5

As you can imagine things can go very wrong if you enter the wrong routing rules e.g. send packets down the wrong network. To ensure that "lost" packets are removed from the network we use a field in the IP header that you will recognise from the first lab: time to live (TTL). This defines the maximum "time" a datagram is allowed to remain on the network i.e. when this counter reaches zero the packet must be dropped. This field technically represents time in seconds, but when each networking element e.g. host or router, forwards a packet it must decrement the TTL count by at least 1, so in reality its a max hop counter.

To simulate a lost packet owing to incorrect routing rules, rather than updating the routing tables on the Pi we can update the MikroTik router's rules. To configure this router you can either use its web

interface via a browser or you can SSH/Telnet into it and use a command line interface. For simplicity we will be using its web interface. In the Pi-1 VNC remote desktop, open a web browser and connect to the router's IP address, as shown in figure 22.

`http:// 172.16.X.6 (where X=Box/Desk)`

password is: 12345. If needed click on the `WebFig` tab. To add a new routing rule click on the left hand panel, selecting:

`IP -> Routing`

Then click on the Add New button and enter the routing rule shown in figure 22 and click OK.

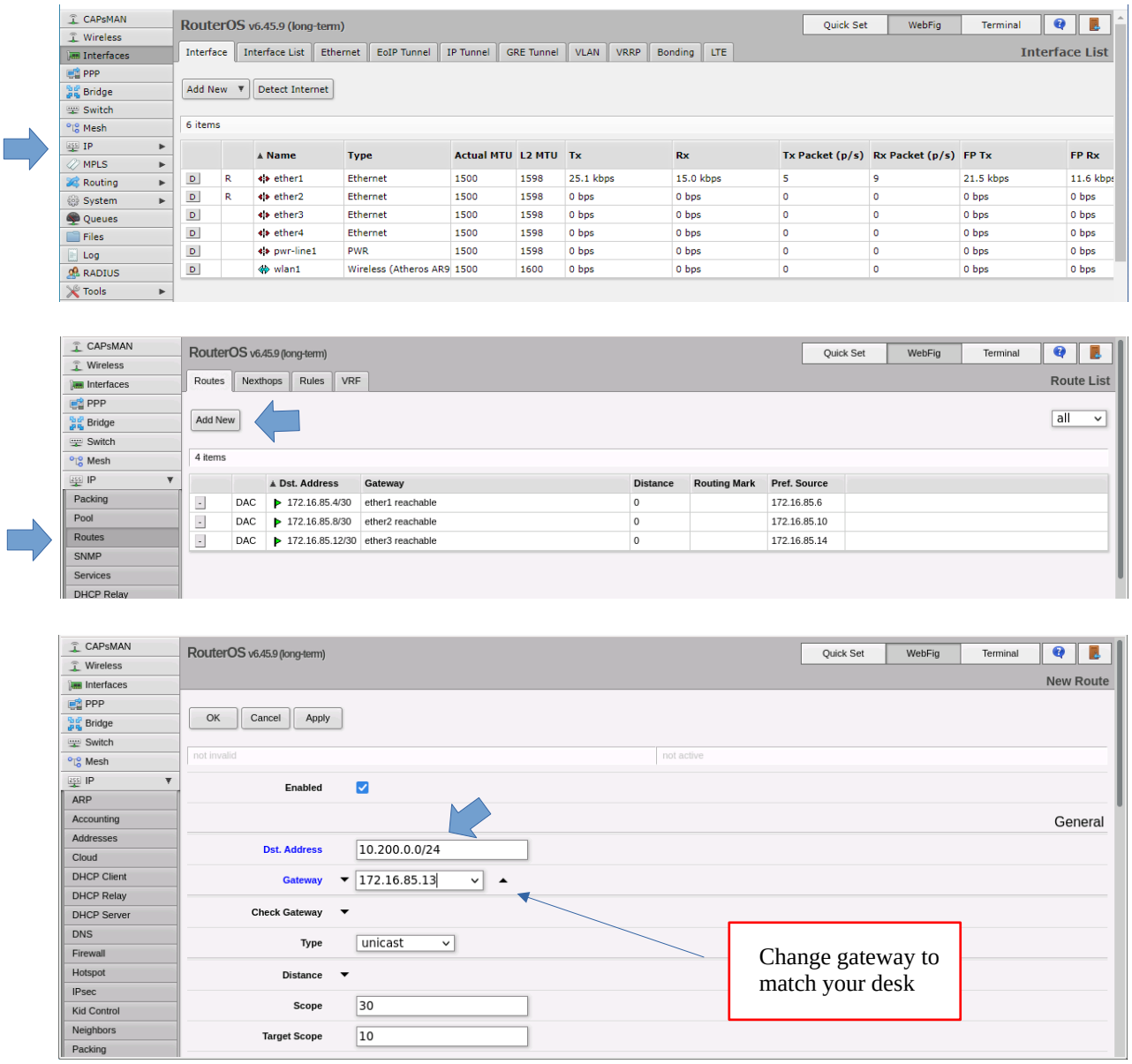


Figure 22 : mikroTik web interface

IMPORTANT, remember to change the gateway IP address to match your desk i.e. the 172.16.X.13 address. The “misconfigured” network 10.200.0.0/24 is common to all desks i.e. an unreachable network, however, any unused private network address would also work.

This routing rule will cause the Pi system to incorrectly route packets to 10.200.0.1, such that it will burn through its TTL count until it is removed.

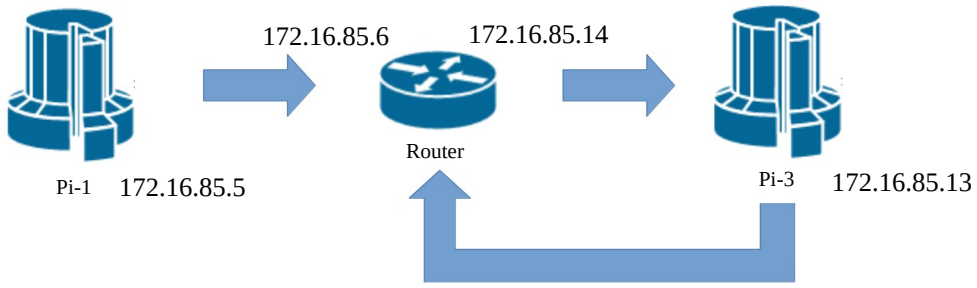


Figure 23 : network loop

We have now setup the network loop shown in figure 23. A set of rules where a packet is mistakenly passed repeatedly from router to router / host i.e. the packet never reaches its destination IP address.

Task : examine the routing tables in figures 24 - 26. Can you see why pinging IP address 10.200.0.1 from Pi-1 cause this IP datagram will get “stuck” in a loop between Pi-3 and the MikroTik router?

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.16.85.6     0.0.0.0         UG    0      0        0 eth0
10.85.1.0       0.0.0.0         255.255.255.0   U     0      0        0 tun0
10.85.2.0       0.0.0.0         255.255.255.0   U     0      0        0 tun0
10.85.3.0       0.0.0.0         255.255.255.0   U     0      0        0 tun0
172.16.85.4     0.0.0.0         255.255.255.252 U     0      0        0 eth0
192.168.0.0     0.0.0.0         255.255.0.0     U     0      0        0 eth1
```

Figure 24 : example Pi-1 routing table

```
 #      DST-ADDRESS        PREF-SRC        GATEWAY            DISTANCE
 0 A S  10.200.0.0/24                      172.16.85.13           1
 1 ADC  172.16.85.4/30     172.16.85.6     ether1                 0
 2 ADC  172.16.85.8/30     172.16.85.10    ether2                 0
 3 ADC  172.16.85.12/30    172.16.85.14    ether3                 0
```

Figure 25 : example Mikrotik router routing table

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.16.85.14    0.0.0.0         UG    0      0        0 eth0
10.85.1.0       0.0.0.0         255.255.255.0   U     0      0        0 tun0
10.85.2.0       0.0.0.0         255.255.255.0   U     0      0        0 tun0
10.85.3.0       0.0.0.0         255.255.255.0   U     0      0        0 tun0
172.16.85.12    0.0.0.0         255.255.255.252 U     0      0        0 eth0
192.168.0.0     0.0.0.0         255.255.0.0     U     0      0        0 eth1
```

Figure 26 : example Pi-3 routing table

To see this loop in action, on Pi-1 set the Wireshark interface to `eth0` and update the packet protocol filter to :

```
!tcp
```

then click on the Apply button, this should remove most of the network noise. Next, click on the Start icon on the top toolbar in Wireshark. Then in the Pi-1 SSH terminal run the following command:

```
ping -c 5 10.200.0.1
```

the `-c 5` option limits the number of pings to 5. If the Mikrotik has been setup correctly you will see the error message “Time to live exceeded”, in the SSH terminal, as shown in figure 27.

```
pi@pi-1:~ $ ping -c 5 10.200.0.1
PING 10.200.0.1 (10.200.0.1) 56(84) bytes of data.
From 172.16.85.13 icmp_seq=1 Time to live exceeded
From 172.16.85.13 icmp_seq=2 Time to live exceeded
From 172.16.85.13 icmp_seq=3 Time to live exceeded
From 172.16.85.13 icmp_seq=4 Time to live exceeded
From 172.16.85.13 icmp_seq=5 Time to live exceeded

--- 10.200.0.1 ping statistics ---
5 packets transmitted, 0 received, +5 errors, 100% packet loss, time 10ms

pi@pi-1:~ $
```

Figure 27 : ping 10.200.0.1

Finally, click on the Stop icon on the top toolbar to stop capturing network traffic. You should now see these ICMP error packets, as shown in figure 28.

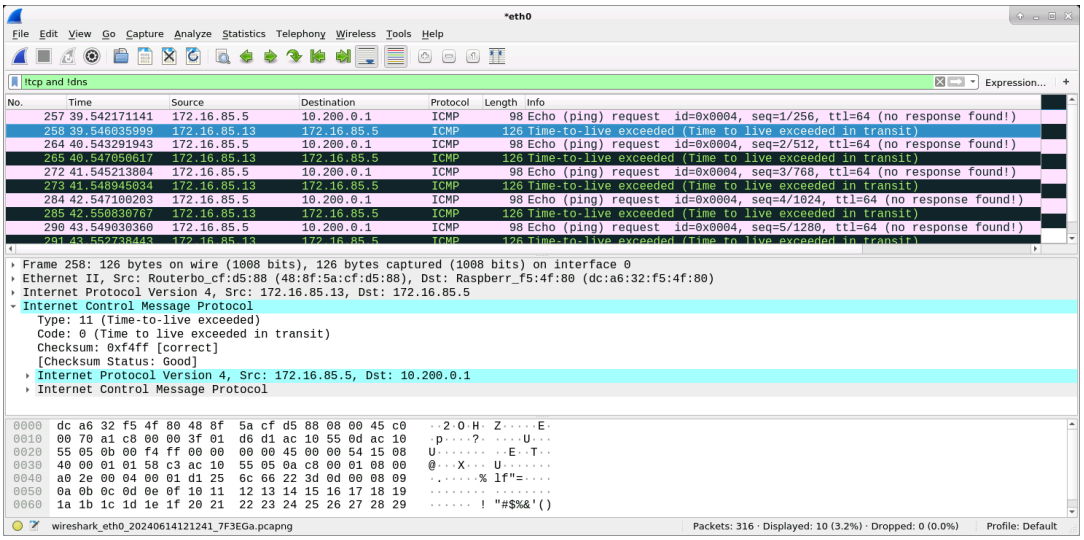


Figure 28 : packet trace for ping 10.200.0.1

We will be looking at the ICMP protocol in a later lecture, but for the moment consider the **BLACK** Time-to-Live exceeded packets shown in figure 28 as error messages from the router letting the Pi

know that something has gone wrong. You can also view the affect of this routing rule by using the `traceroute` command. In the Pi-1 SSH terminal run the following command:

```
traceroute 10.200.0.1
```

Task : Do you understand why the strange trace route output looks the way it does :).

IMPORTANT : when you have finished, restore the Mikrotik configuration back to its original settings, such that it's good to go for the next person using that desk. If you are not sure how to do this please do ask and I can shown you how.

Appendix A : routing rule

```
sudo ip route add 10.X.1.0/24 dev tun0          (where X=desk)
```

Appendix B : tunnel tun1

Pi-2

```
sudo ip tunnel add tun1 mode ipip local 172.16.X.9 remote 172.16.X.13
sudo ip link set tun1 up
sudo ip addr add 10.X.3.3/24 dev tun1
```

Pi-3

```
sudo ip tunnel add tun1 mode ipip local 172.16.X.13 remote 172.16.X.9
sudo ip link set tun1 up
sudo ip addr add 10.X.3.2/24 dev tun1
```

Appendix C : routing rules

Pi-2

```
sudo ip route add 10.X.3.0/24 dev tun1
```

Pi-3

```
sudo ip route add 10.X.2.0/24 dev tun1
```

Appendix D : routing rules

Pi-1

```
sudo ip route add 10.X.3.0/24 dev tun0
```

Pi-3

```
sudo ip route add 10.X.1.0/24 dev tun1
```

Appendix E : setup scripts

To speed up the construction of these tunnels you can download and use the scripts `pi-1.sh`, `pi-2.sh` and `pi-3.sh` which are available on the VLE in the file: `scripts.zip`.

Edit these scripts to remove any commands that you have already entered e.g. for `pi-1` or `pi-2`. Then copy these files to the appropriate Raspberry Pi e.g. `pi-3.sh` to `pi-3` etc. To run each script in a terminal you will first need to make it executable by running the command:

```
chmod +x pi-X.sh                    (where X = 1,2 or 3)
```

Then run each script by entering the command:

```
./pi-X.sh                           (where X = 1,2 or 3)
```

pi-1.sh

```
#!/bin/sh

sudo ip tunnel add tun0 mode ipip local 172.16.X.5 remote 172.16.X.9
sudo ip link set tun0 up
sudo ip addr add 10.X.1.2/24 dev tun0

sudo ip route add 10.X.2.0/24 dev tun0
sudo ip route add 10.X.3.0/24 dev tun0
```

pi-2.sh

```
#!/bin/sh

sudo ip tunnel add tun0 mode ipip local 172.16.X.9 remote 172.16.X.5
sudo ip link set tun0 up
sudo ip addr add 10.X.2.2/24 dev tun0

sudo ip tunnel add tun1 mode ipip local 172.16.X.9 remote 172.16.X.13
sudo ip link set tun1 up
sudo ip addr add 10.X.3.3/24 dev tun1

sudo ip route add 10.X.1.0/24 dev tun0
```

pi-3.sh

```
#!/bin/sh

sudo ip tunnel add tun1 mode ipip local 172.16.X.13 remote 172.16.X.9
sudo ip link set tun1 up
sudo ip addr add 10.X.3.2/24 dev tun1

sudo ip route add 10.X.2.0/24 dev tun1
sudo ip route add 10.X.1.0/24 dev tun1
```