Oscar Esparza
Jose L. Muñoz
Juanjo Alins
Jorge Mata

UPC Telematics Department

IP Tunnels

Contents

0.1 Introduction

According to wikipedia, “An IP tunnel is an Internet Protocol (IP) network communications channel between two networks.” So, we are going to create channels between networks to transport protocols that, generally, are not easily routed in an intermediate transport network. There are many uses of tunneling techniques, for instance the creation of virtual private networks (VPNs), or the progresive deployment of new technologies by connecting islands that use the same protocol (multicast, IPv6). IP tunneling uses encapsulation of packets, that is, every IP packet, including addressing information of its source and destination IP networks, is encapsulated within another packet format native to the transit network. Mobile IP is also one of the most prominent uses of encapsulation, as a way to deliver datagrams from a mobile node’s "home network" to an agent that can deliver datagrams locally to the mobile node away from home.

There are many types of tunnels, using different protocols and layers, for example: IPIP, GRE or IPSec at the network layer, SSL or TLS at the transport layer, or SSH at the application layer. In this lab session, we are going to use the most simplistic IP tunnel, IPIP, sometimes also called `ipencap` or simply `encap`. When using IPIP, the protocol used in the underlying network is IP (tunneling protocol), and the tunneled protocol (encapsulated protocol) is also IP. This protocol is described in RFC 2003, which is mandatory to read and understand prior starting with this lab sessions. We will use the same nomenclature of this RFC to identify the different components of the IP tunnel:

```
 << ... In the most general tunneling case we have

   source ---> encapsulator --------> decapsulator ---> destination

with the source, encapsulator, decapsulator, and destination being
```

```
separate nodes.  The encapsulator node is considered the "entry point"
of the tunnel, and the decapsulator node is considered the "exit point"
of the tunnel ... >>
```

To encapsulate an IP packet in another IP packet, an outer header is added. In this new header, the source IP address is the encapsulator (the entry point of the tunnel), and the destination IP address is the decapsulator (the exit point of the tunnel). The inner packet (and inner header) remains mostly unmodified, as it is consider data.

In this lab session, we pretend to explain some typical problems that appear when using tunneling techniques and encapsulation. Some of these problems are related to how network error messages are treated inside of tunnels (`ICMP Relaying` and `soft state`), how fragmentation is managed in the encapsulator/decapsulator, and how can we avoid some problems that appear due to the use of TCP inside of tunnels. For this reason, it is important to fully know and understand the fields of the IP header, emphasizing in those aspects related to fragmentation (if not, see RFC 791). Also, it is also mandatory to know how a TCP connection is established and fully understand the concept of MSS (if not, see RFCs 793 and 879). Knowledge about filtering rules (`iptables`) and netcat (`nc`) will be also necessary.

This lab practice has been designed to last two sessions (4 hours), that should be done in the lab room so the professor can help you to understand the key points. There are also some additional exercises that must be at home (labelled as "TO DO AT HOME"), just to practise some side aspects about tunnels.

0.2 The scenario of the lab sessions

The aim of this exercise is to get in touch with *IP tunneling* techniques. To do so, we are going to use a tipical example of use of IP tunnnels, the connection of remote private IP networks using an intermediate public network. The scenario of this laboratory session is depicted in Figure 1. The private networks (**SimNet0**&**SimNet3**) represent remote branches of a certain business. We will connect these remote branches using IP tunnels. The public networks (**SimNet1**&**SimNet2**) that connects these branches represents the Internet.

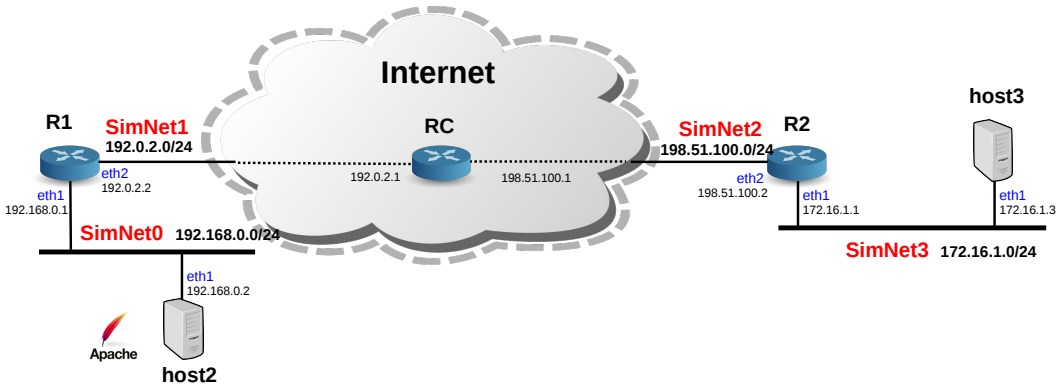

Figure 1: IP tunnel scenario

0.2.1 The branches

We will assign the following IP addresses to the different branches:

Branch	IP Network	Public or private
SimNet3	172.16.1.0/24	private addressing
SimNet0	192.168.0.0/24	private addressing

The routers **R1**, **R2** use SNAT (Source NAT) to allow their internal users to access to the external hosts. At the beginning of the lab session we are not going to use any filtering rule, but later we will include some `iptables` rules to filter traffic. Remember that you should know how to deal with filtering and NAT rules (with `iptables`) to make this lab session. If not, you should review these concepts prior to starting this practice.

0.2.2 The Internet

We have assumed a very simplistic Internet with only one router, **RC**. These are the public addresses assigned for the Internet:

Branch	IP Network	Public or private
SimNet1	192.0.2.0/24	public addressing
SimNet2	198.51.100.0/24	public addressing

So, the problem in this scenario is that IP packets between branches cannot be directly routed through the Internet because of private adressing. For this reason, we are going to configure an IPIP tunnel between the edge routers to make IP packets be routed through the Internet.

0.3 Starting the scenario using simctl

First of all, we will start the virtual scenario with the command:

```
host$ simctl iptunnel start
```

When starting this scenario, IP addresses in network interfaces and routing tables should be properly configured. If not, maybe there is a problem in the scenario or in `simctl`, so contact the professor.

0.4 Configuring IPIP tunnels

Now we are going to configure the tunnel of the scenario. Several commands should be executed in the border routers (encapsulator y decapsulator) to properly configure a tunnel. In particular, in each border router you have to:

- Configure the tunnel interface by means of the command `ip tunnel` (consult the man page of this command). We will start configuring the tunnels to work in **ipip** mode, and with the option **nopmtudisc** set to avoid the algorithm of Path MTU Discovery. Notice that this latest option force the tunnel to also set the option **ttl inherit**. As an example, in **R2** you have to execute:

```
R2$ ip tunnel add tunnel0 mode ipip local 198.51.100.2 remote 192.0.2.2 ttl 0 nopmtudisc
dev eth2
```

- Activate the new virtual interface of type tunnel. To do so, you have to assign an IP address to it (for instance, by means of the `ifconfig` command). In linux, it is necessary to assign an IP address to any interface if we want it to properly work. In this lab session, we are not going to use this address, so no matter which address is configured. Just as an example:

```
R2$ ifconfig tunnel0 1.2.3.4
```

- Update the routing table to make the proper packets to be routed towards the tunnel (for instance, by means of the `route` command). Just as an example:

```
R2$ route add -net 192.168.0.0/24 dev tunnel0
```

Now that you know how to do it, create the IPIP tunnel between **R1** and **R2**. Remember that it is better to store in a file all the commands used to configure the system, so you can reuse them later.

0.5 Testing the tunnels: protocol and TTL fields

Once this configuration has been applied in the edge routers, we will test if the tunnel is working properly:

1. Open FOUR `wireshark` network analyzers in the PHYSICAL HOST and capture traffic in each of these networks: **SimNet0**, **SimNet1**, **SimNet2** and **SimNet3**.
   To properly see fragmented messages, DISABLE the following option in all these wiresharks:
   `Edit-Preferences-Protocols-IPv4-Reassenble fragmented IPv4 datagrams.`

2. Send the following `ping` from **host2** to **host3**:

```
host2$ ping -c1 172.16.1.3
```

   (a) What is the meaning of the `-c` option? (consult the man page of ping if necessary)

   (b) Is the ping working? How many packets can you see?

   As you can see, the `-c` option allows us to configure the number of ICMP *echo-request* messages sent, only one in this particular case. If you properly made the configuration of the tunnel, the ping should work. If the ping is not working, review your configuration until it works.

3. Assuming that the ping is working, it is necessary to emphasize certain aspects related to tunneling issues:

   (a) Verify that the IP packet in **SimNet1** and **SimNet2** is encapsulated using IPIP. How many IP headers can you see in these packets? Which is the value of the "Protocol" field in the Outer Header?

   (b) Determine the difference in size from the original IP packet (in **SimNet0**) and the encapsulted one (in **SimNet1**).

   (c) Identify the value of the "TTL" field in the original IP packet and how this value evolve during the transmission, both in the inner and in the outer headers. Remind that we used the option `ttl inherit` during the creation of the tunnels.

TTL	SimNet0	SimNet1	SimNet2	SimNet3
Outer header				
Inner header				

4. To evaluate the behavior of the tunnel when the TTL expires, we will send IP packets from **host2** (192.168.0.2) to **host3** (172.16.1.3) increasing one by one the TTL. First, clear and start again the capture in the four wiresharks to perfectly monitor step by step the transmission of this packets in all the involved networks (**SimNet0**, **SimNet1**, **SimNet2** and **SimNet3**). Next, send again one ICMP *echo-request* messages using the `ping` program and TTL=1.

```
host2$ ping -c1 -t1 172.16.1.3
```

   (a) Can you see packets in all the networks? Did the packet reach the destination host? In which host the packet was lost?

   (b) Can you see any ICMP error message? Which one? In which network?

(c) Which are the origin and destination IP addresses of this error message?

Now clear all the wiresharks and send again another ping, but with TTL=2.

(a) Did the packet reach the destination host? In which machine the packet was lost?

(b) Can you see any ICMP error message? Which one? In which network?

(c) Which is the origin and destination IP addresses of this error message?

(d) Can you see any ICMP relaying?

(e) Send again the same ping with TTL=2. Is there any `soft state`?

(f) Is this behavior compliant with RFC 2003 (see section 4.4)?

Now use TTL=3:

(a) Did the packet reach the destination host?

(b) Can you see any ICMP reply message?

0.6 Testing the tunnels: nopmtudisc

Now we will practise how *IP Fragmentation* works when combined with IP tunnels when using the `nopmtudisc` option. Note that fixed ttl is incompatible with this option: tunnels with fixed ttl always make pmtu discovery. Remember that the maximum length of an IP datagram is 64KBytes. This maximum lenght is generally not reached, as most transmission links have smaller maximum packet length limits. The *Maximum Transmission Unit* (MTU) is defined as the size of the largest IP packet (including the IP header) that the link layer can transmit. The value of the MTU depends on the type of transmission link, for example, ethernet links have a MTU of 1500 bytes.

IP routers can fragment IP datagrams, and the receiving station is responsible for reassembling the fragments. Fragmentation consists in breaking the datagram into pieces that can be reassembled later. The main fields of the IP header involved in fragmentation are “identification”, “total length”, and “fragment offset”. There are also the "more fragments" (MF) and "don’t fragment" (DF) flags. There are several issues that make IP fragmentation undesirable, mainly increasing CPU and memory overhead to fragment/reassemble IP datagrams in sender, routers and receivers. For instance, reassembly is very inefficient in routers, whose primary job is to forward packets as quickly as possible. Also, handling dropped fragments is quite complicated for routers. We will assume that you know how fragmentation and reassembly works, but if not read RFC 791.

The "Path MTU" is the smallest MTU of all the hops of the path between source and destination. This path MTU is the largest packet size that can traverse this path without suffering fragmentation. Path MTU Discovery (described in RFC 1191) describes the technique to discover the path MTU between two IP hosts. Basically, the DF (don’t fragment) flag is set in the IP headers of outgoing packets, so any router in the path whose MTU is smaller than the packet will drop it, sending back an ICMP *Destination Unreachable - Datagram Too Big* message (also known as *Fragmentation Needed*). This error message contains the value of this more restrictive MTU, and it is used by the source host to reduce its assumed path MTU. If the process is repeated until the packet reaches destination, the packet size is the path MTU. As we will see in this lab session, this PMTUD technique may present problems in case of using IP tunnels, and also when there are firewalls that filter ICMP packets.

0.6.1 Limiting the MTU

1. Use `ifconfig` to see which is the value of the MTU in the tunnel inferfaces both in **R1** and **R2**.

   (a) Which is this value?

   (b) Is there any relationship between the MTU assigned to the tunnel, and the MTU assigned to the physical interface?

2. Delete the previously configured tunnel in **R2**. Instead of doing it manually, you can use this script to do so. Go to the console of your HOST machine, and execute:

```
HOST$ simctl iptunnel exec deltun
```

   This will delete the tunnel in both sides **R1** and **R2**.

3. Change the MTU of **SimNet2** to the value 996 bytes. So, get the consoles of **R2** and RC and use the `ifconfig` command to do so. For instance, in **R2** execute:

```
R2$ ifconfig eth2 mtu 996
```

4. Reestablish again the tunnel using the following script:

```
HOST$ simctl iptunnel exec addtun_nopmtu
```

   (a) Which is now the MTU in both sides of the tunnel?

   (b) Why this change in the MTU value?

0.6.2 Fragmentation fields and options

We pretend to see how the fragmentation fields of the Outer header depend on the Inner header and also on the configuration of the tunnel.

1. Put all wiresharks listening traffic in all SimNet interfaces.

2. Send the following `ping` from **host2** to **host3**:

```
host2$ ping -c1 -s 500 -M want 172.16.1.3
```

   Consult the man page of `ping` to know what is the meaning of the `-s` and `-M` options.

   (a) How many ICMP packets can you see? What kind of packets?

   (b) Which is the size of these packets? Describe the headers of these packets.

   (c) Have these packets been fragmented? Why?

   (d) Which is the value of the "don't fragment" (DF) flag in the ICMP packets in **SimNet0**?

   (e) And in **SimNet1**, which is the value of DF in the Outer and Inner headers?

   After this brief test, you should have noticed that the `-s` option is used to set the number of bytes in the "data" section of ICMP. The `-M` option is used to manage fragmentation in the origin host. The `-M want` option is intended to "do PMTU discovery, fragment locally when packet size is large". That is, the DF flag is set by default, but when there is the necessity of fragmentation, the origin can do it.

3. Now execute:

```
host2$ ping -c1 -s 500 -M dont 172.16.1.3
```

   (a) What is the difference compared to the previous test?
   As you can see, the `dont` option ("do not set DF flag") allows fragmentation in path.

4. Now execute:

```
host2$ ping -c1 -s 500 -M do 172.16.1.3
```

(a) Can you see any difference in wireshark compared to the first case? Is there any difference at all?
The `do` option "prohibit fragmentation, even local one". In this previous case you will see no difference, but later in other tests there can be some differences.

Now that you understand how encapsulation is performed, calculate which is the size necessary to make the encapsulated packet have the maximum sized alowed by **SimNet2**. Send a ping of this size and verify that there is no fragmentation.

The `-M want` option

We start the test with the `-M want` option because it is the default option used by ping. Now you are going to cause packets to be fragmented, and to see how the encapsulator (**R1**) behaves in relation to the management of messages *ICMP Datagram Too Big* (Type 3, Code 4, also known as *Fragmentation Needed*), used in the PMTU Discovery mechanish.
It is important to know that these messages are able to modify the status of the dynamic routing table (*kernel routing cache*). Sometimes during the lab session you will be asked to delete this kernel routing cache, so the starting point of the test will be always the same. This can be done by manually executing the following command in all the involved hosts and routers:

```
$ ip route flush cache
```

However, we recommend to execute the following script, that automatically deletes this cache in all machines involved:

```
HOST$ simctl iptunnel exec flushcache
```

So, you are going to increment the size of packets to cause fragmentation:

1. Delete the kernel routing cache executing the label `flushcache`.
2. Put all wiresharks listening traffic in all SimNet interfaces.
3. Execute the following command in **host2**:

   ```
   host2$ ping -c1 -s 1000 -M want 172.16.1.3
   ```

   (a) Is the ping working?
   (b) How many ICMP packets can you see in **SimNet0**? What kind of packets? Is set the DF flag in the IP header of the ICMP *echo-request*?
   (c) And in **SimNet1**, what packets can you see? Is the DF flag set in the IP headers (inner and outer)?
   As you can see, there is an error produced by the MTU by means of a message ICMP *fragmentation-needed*. This error apears due to the DF flaf is Set in the Outer IP header of the ICMP *echo-request* message.
   (d) What device is the origin of the ICMP *fragmentation-needed* message? And the destination? Which is the maximum MTU notified in this message?
   (e) In your opinion, what entity should be the proper destination of this message?
   (f) According to RFC2003 (see Section 4.1), is **R1** acting as relay of the message and sending it to **host2**?

As you can see, in the case of *fragmentation-needed* messages, the encapsulator is not performing `ICMP Relaying`. Instead, the encapsulator uses `soft state` to inform the origin host about the error. To show this, we are going to send more than one message to use the kernel routing cache:

1. Delete the kernel routing cache executing the label `flushcache`.

2. Put all wiresharks listening traffic in all SimNet interfaces.

3. Execute the following command in **host2**:

```
host2$ ping -c2 -s 1000 -M want -i1 172.16.1.3
```

With this command we are sending two ICMP `echo-request` messages with 1000 Bytes of data and separated by an interval (-i option) of 1 second.

(a) Is the ping working? How many ICMP messages can you see in **SimNet0**? Is the DF flag set in the IP header of these messages?

(b) In **SimNet1**, how many ICMP *echo-request*s can you see? What about the DF flag?

(c) Can you see an ICMP *fragmentation-needed* message in **SimNet0**? Which is the origin and destination of this message? Which is the maximum MTU notified in this message?

(d) Do you think that **R1** maintains a `soft state` of the MTU of the tunel?

This `soft state` can be seen by executing the following command in **R1**:

```
R1$ ip route show cache
...
172.16.1.3 dev tunnel0  src 192.0.2.2
    cache  expires 596sec ipid 0xf9ba mtu 976
...
```

Notice that the information contained in this routing cache is ephemeral (10 min). If you lasted more than this time to execute the previous command maybe you will find the cache empty, so you have to repeat the experiment.

Now, we will send three ICMP *echo-request* messages:

1. Delete the kernel routing cache executing the label `flushcache`.

2. Put all wiresharks listening traffic in all SimNet interfaces.

3. Execute the following command in **host2**:

```
host2$ ping -c3 -s 1000 -M want -i1 172.16.1.3
```

Now let us analyze this test in detail:

(a) Is the ping working? How do you know that?

(b) In **SimNet0**, have a look at the DF flag in all the ICMP *echo-request* messages. What can you deduce? Is there any difference in the third ICMP Echo request message?

(c) In **SimNet0**, who is performing the fragmentation of the third ICMP *echo-request*? Which are the sizes of the packets that compose this third ICMP *echo-request*? Take notes of the existing headers.

(d) In **SimNet1**, how many fragmented packets can you see that correspond to this third ICMP *echo-request*? What about the DF flag in the Inner and Outer IP headers?

(e) Describe in detail the sizes and headers of these packets. Is fragmentation performed in the same way than **SimNet0**? Why the difference? Notice that the Fragment Offset field in the IP header is meassured in units of 8 bytes.

(f) In **SimNet3**, how many ICMP *echo-request*s can you see? Describe sizes and headers.

(g) In **SimNet3**, how many ICMP *echo-reply* messages can you see? Describe sizes and headers.

(h) In **SimNet2**, how many fragmented packets can can you see that correspond to this third ICMP *echo-reply*? Describe in detail sizes of these packets and headers. Is fragmentation performed in the same way than for the ICMP Request message? Describe in detail sizes of these packets and headers.

Finally, we will send the latest ping of this series:

1. Delete the kernel routing cache executing the label `flushcache`.
2. Put all wiresharks listening traffic in all SimNet interfaces.
3. Execute the following command in **host2**:

```
host2$ ping -c2 -s 1460 -M want -i1 172.16.1.3
```

(a) Is there any error message? Who is the sender of this error?

(b) Which is the value of MTU that is reporting this error message? Is this value the real path MTU value? Why there are no more errors during the transmission? Have a look to the DF flag.

The `-M do` option (TO DO AT HOME)

Now, we are going to test the `-M do` option. Remember that the `-M do` option "prohibit fragmentation, even local one". Let us see if there is any difference with the previous `-M want` option.

1. Delete the kernel routing cache executing the label `flushcache`.
2. Put all wiresharks listening traffic in all SimNet interfaces.
3. Execute the following command in **host2**:

```
host2$ ping -c3 -s 1000 -M do -i1 172.16.1.3
```

(a) Is the ping working? How many ICMP *echo-request* packets can you see in **SimNet0**? Is there any fragmented packet?

(b) Can you see the difference with the previous `-M want` option?

The `-M dont` option (TO DO AT HOME)

Now, we are going to test the `-M dont` option. Remember that the `-M dont` option ("do not set DF flag") allows fragmentation in path.

1. Delete the kernel routing cache executing the label `flushcache`.
2. Put all wiresharks listening traffic in all SimNet interfaces.
3. Execute the following command in **host2**:

```
host2$ ping -c3 -s 1000 -M dont -i1 172.16.1.3
```

(a) Is the ping working? Is there any error message? Why? Have a look to the DF flags.

(b) According to all the previous results, which is the best option to use? Think about pros and cons of fragmentation in path.

0.7 Testing tunnels: pmtudisc (TO DO AT HOME)

All previous test were performed using the nopmtudisc option when creating the tunnel. Let us change this option to see if there is any difference.

1. Delete the previously configured tunnel:

```
HOST$ simctl iptunnel exec deltun
```

2. Reestablish again the tunnel but now with the `pmtudisc` option enabled:

```
HOST$ simctl iptunnel exec addtun_pmtu
```

   Now, verify the configuration in both edge routers with this command:

```
$ ip tunnel show
```

   Also, verify that the MTU in both sides of the tunnel is the proper one.

Now, do the following test:

1. Delete the kernel routing cache executing the label `flushcache`.
2. Put all wiresharks listening traffic in all SimNet interfaces.
3. Execute the following command in **host2**:

```
host2$ ping -c3 -s 1000 -M want -i1 172.16.1.3
```

   (a) Is the ping working? How many ICMP `echo-request` can you see in **SimNet0**? Is there any error message? In which network?
   (b) Try to explain the behavior of the encapsulator.

Now, let us test with the `-M do` option:

1. Delete the kernel routing cache executing the label `flushcache`.
2. Put all wiresharks listening traffic in all SimNet interfaces.
3. Execute the following command in **host2**:

```
host2$ ping -c3 -s 1000 -M do -i1 172.16.1.3
```

   (a) What is the difference regarding the previous test?

Finally, let us test the latest `-M dont` option:

1. Delete the kernel routing cache executing the label `flushcache`.
2. Put all wiresharks listening traffic in all SimNet interfaces.
3. Execute the following command in **host2**:

```
host2$ ping -c3 -s 1000 -M dont -i1 172.16.1.3
```

   (a) Is the ping working? How many ICMP `echo-request` can you see in **SimNet0**? Is the DF Set in the packets in **SimNet0**?
   (b) In **Simnet1**, can you see ICMP `echo-request` packets? Which one, the first, second or third?
   (c) Is there any error message in **SimNet1**? Why this error? Have a look to the DF flag in inner and outer header `echo-request`.
   (d) Why the encapsulator is not sending the rest of pings towards the tunnel? Consult the kernel routing cache.

0.8 Testing tunnels: TCP and MSS

Now you are going to test how the TCP protocol behaves when the connection go through a tunnel. First, reestablish the tunnel with the option `nomptudisc`. Leave the MTU=996 in **SimNet2**.

```
HOST$ simctl iptunnel exec deltun
HOST$ simctl iptunnel exec addtun_nopmtu
```

Remember that the *Maximum Segment Size* (MSS) is the maximum amount of data (in bytes) that TCP can receive in a TCP segment. In the 3-way handshake (SYN, SYN/ACK, ACK) of TCP, each side informs its MSS value to the other side. This is done by means of the "Options" field of the TCP headers in the SYN segments. Each host calculates its MSS by deducting the minimum IP and TCP headers to the MTU value. In case of using Ethernet, a host will advertise a MSS of 1460 bytes (1500 bytes of MTU - 20 bytes of minimum IP header - 20 bytes of minimum TCP header). With this information, the sending host is required to limit the data inside a TCP segment to a value less than or equal to the MSS reported by the receiving host. Notice that MSS only has sense when using the TCP protocol, and also that this is not a negotiation, the MSS is advertised to the other side. In addition, the MSS is a fixed value for each host, and it only depends on the MTU of the link layer interface of that host. Despite both sides of the TCP connection (client and server) are independent and may have different MSS, due to symmetry, any sender will use the lower of the two values.

From previous tests, you know that the use of tunnels causes problems due to the addition of tunnel headers. Most of these problems are solved when hosts use PMTUD and receive ICMP *fragmentation-needed* messages, so they can fragment packets in origin.

The problem occurs when the system administrator of a network decides to filter ICMP messages to avoid security attacks. This is more common than you think, and produces weird effects on TCP connections. For this reason, we are going to reproduce this scenario here in the lab.

To continue with this test, remember that you must be familiar with these contents:

- TCP: 3-way handshake, MSS concept.
- Use of netcat (`nc` command) to establish TCP connections.
- Basic concepts of Linux operating systems: redirection operators (>, <, |, »)
- Filtering rules with `iptables`.

0.8.1 Netcat and TCP

First, we will start doing a correct transmission (without filtering rules) using netcat:

1. Put all wiresharks listening traffic in all SimNet interfaces.
2. Start a netcat server in **host3** listening in port 12345 (the `-l` option means `listen`, that is, this host is behaving as server):

   ```
   host3$ nc -l -p 12345
   ```

3. Start the netcat client in **host2**.

   ```
   host2$ nc 172.16.1.3 12345
   ```

   Netcat is used to establish a TCP connection between **host2** and **host3**. As you can see, the tool behaves like a chat, and if you type something on the keyboard of one host, it appears in the other side. If you want to close netcat, you can type `Ctrl+C`, and the TCP connection will be closed. Have a look to all wiresharks and answer:

   (a) Can you see the classical 3-way handshake of TCP?

   (b) Which is the value of the MSS (Maximum Segment Size) advertised in this handshake?

   (c) What is the meaning of this MSS? Can have the client a MSS different from the server?

0.8.2 Netcat to transfer files

Netcat can also be used to transfer files:

1. Delete the kernel routing cache executing the label `flushcache`.
2. Put all wiresharks listening traffic in all SimNet interfaces.
3. Start the netcat server in **host3**, that will send the file `/etc/services` towards the client when the connection has been established:

```
host3$ nc -l -p 12345 -q 0 </etc/services
```

4. Start the netcat client in **host2**, that will display in the screen the file `/etc/services` of the server.

```
host2$ nc 172.16.1.3 12345
```

   (a) In the 3-way handshake, which was the value of the MSS?
   (b) Did you notice any problem during the transmission according to the wiresharks?
   (c) Can you see any ICMP message in **SimNet3**? Who is sending these error messages and why?
   (d) Was the file transmitted properly? Who solved the problem?
   (e) Is this meaning that the MSS has changed during the transmission?

As you can see from the previous example, TCP is a reliable protocol. This means that it will retransmit lost information until it is properly received (or after a certain number of unsuccessful retransmissions). TCP is able to change the lenght of the segments in case that the network reports problems of size, but this does not mean that the MSS changes, as it is a fixed value.

Obviously, files can be transmitted in both directions using netcat. In the previous example, we made a transmission in which the server transferred a file towards the client. Now we will test this same operation in the reverse direction.

1. Delete the kernel routing cache executing the label `flushcache`.
2. Put all wiresharks listening traffic in all SimNet interfaces.
3. Start the netcat server in **host3**:
4. Start the netcat client in **host2**, but in this case inject the file `/etc/services` of the client towards the server using the TCP connection.

   (a) Which are the commands that you used to do the previous behavior?
   (b) Did you notice any problem during the transmission according to the wiresharks?
   (c) Can you see any ICMP message in **SimNet0** and **SimNet1**? Who is sending these error messages and why?
   (d) Was the file transmitted properly? Who solved the problem?
   (e) According to the transmission, who is the TCP client and who is the TCP server?

   Again, TCP solved the problem by adapting the lenght of the segments to the network conditions and retransmitting lost information. Notice that the TCP client is the host that started the TCP dialogue by sending the first message SYN, and the TCP server is the host that answered by sending the SYN-ACK message, no matter of the direction of the data transmission.

0.8.3 TCP, tunnels and filtering

A problem that appears in actual networks is that some networks administrators filter all traffic that they suppose to be dangerous. In this case, it is usual to filter almost all UDP traffic (except the DNS) and in general all ICMP traffic.

Now we are going to test what happens when error messages are not able to reach the destination host. In particular, we will see which are the consequences of filtering *Fragmentation Needed* messages.

1. Let us imagine that the network administrator of R1 wants to protect this router against typical attacks that use ICMP. For this reason, it avoids sending ICMP messages to this router. So, execute the following command to filter the ICMP traffic whose destination is **R1**:

```
R1$ iptables -A INPUT -p icmp -j DROP
```

2. Delete the kernel routing cache executing the label `flushcache`.

3. Put all wiresharks listening traffic in all SimNet interfaces.

4. Again using netcat, send the file `/etc/services` from **host2** (acting as client) towards **host3** (acting as server).

   (a) Can you see the 3-way handshake? Did you notice problems with these 3 initial messages?

   (b) Did you notice problems with data segments? Why? Which is the difference with the previous ones?

   (c) Can you see any ICMP message in **SimNet1**? And in **SimNet0**?

   (d) Was the file transmitted properly? Why?

   (e) What will happen if instead of the INPUT chain, we use the FORWARD chain in the filtering rule?

As you can see, the client is not receiving feedback from the errors of the network due to the filtering rule, so it is unable to addapt the TCP transmission to the conditions of the network. TCP assumes that losses are due to congestion, so it will try to retransmit lost segments just waiting more and more time.

0.8.4 Changing the MSS

Obviously, the previous problem can be solved by deleting the `iptables` filter rule in **R1**, but this may be impossible if we do not have the admin rights to do so. As an alternative solution, it is possible to change the advertised MSS to a different value, so tunneled packets do not need to be fragmented in path. So, we have to change the advertised MSS in the 3-way handshake to a different value from the one calculated by default in the host (and which mainly depends on the MTU of the exit network interface).

There are some issues that should be considered prior to doing this change:

- Which is the value of the MSS that avoids fragmentation in path? To calculate this, we should clearly know and understand the notion of MTU, MSS and encapsulation. Remember that the MSS counts only **DATA** octets in the segment, and it does not count the TCP header or the IP header. So, the MSS is calculated as the most restrictive MTU in the path minus the minimum IP, TCP and tunnel headers.

- Which message in the 3-way handshake (SYN or SYN/ACK) should be modified. Remember that the client informs to the server about its MSS using the SYN segment (so this affects data that go from the server to the client), meanwhile the server informs to the client about its MSS using the SYN/ACK segment (data from the client to the server).

- Where we can change this MSS, hosts or routers?

– Changing the MSS in the end systems (origin or destination). In Linux systems it is possible to force that the TCP connections that use a certain route use a particular MSS during the establishment phase. In general, the command is
`ip route add ROUTE advmss VALUE`
where ROUTE is the route we want to add (more information using `ip route show`), and VALUE is the "Advertised MSS" in bytes. In case the route exists, instead of `add` use `change`. Just as an example (not related with this particular case of the lab session), let us imagine that we want to change the MSS that **host2** advertises in the 3-way handshake to the value 1440. In this particular case, the command would be:
`host2$ip route add 172.16.1.0/24 via 192.168.0.1 advmss 1440`

– Changing the MSS in the routers. If the TCP traffic goes through a router that we manage, it is possible to modify the advertised MSS in the 3-way handshake. To do so, we will use `iptables`, and in particular the "mangle" table. This table is used to modify packets, and it may be used with the chains PREROUTING, INPUT, FORWARD, OUTPUT and POSTROUTING. The chain associated to the change of MSS is TCPMSS, and it has two parameters:
`--set-mss value`: to establish the MSS to `value` in bytes.
`--clamp-mss-to-pmtu`: to establish the MSS to the one obtained via PMTUD.
For instance:
`iptables -t mangle -A FORWARD -o tunnel0 -p tcp --syn -j TCPMSS --set-mss 1440`
With this command, the linux router will change the MSS to 1440 bytes to those TCP packets with the SYN flag set (and the rest of flags not set) and that exit the router via the interface `tunnel0`.

Change the MSS in the host

So, we are going to try to solve the previous transmission but changing the MSS in the hosts.

1. Delete the kernel routing cache executing the label `flushcache`.
2. Put all wiresharks listening traffic in all SimNet interfaces.
3. Identify which are the proper parameters to change the mss properly:
   - Which is the value of the advertised MSS to avoid fragmentation in path? Notice that the MTU in **SimNet2**=996.
   - Which message of the 3-way handshake of TCP should be change? Why?
   - If we are changing the MSS in a host, which is the host where we should execute the command?
4. Now that you fully know all the parameters to properly change the MSS, use `ip route` to change it.
5. Using netcat again, send the file `/etc/services` from **host2** (acting as client) towards **host3** (acting as server).
   (a) Did you notice problems with data segments?
   (b) Can you see any ICMP message in **SimNet1** and **SimNet0**?
   (c) Was the file transmitted properly?

Prior to continue with the following test, delete the route that you configured in the host to change the MSS.

Change the MSS in the router

Now we are going to simulate a quite typical problem that appear when doing web browsing and tunnels. There is a web server (`apache2`) listening in **host2**. Notice that when using `simctl`, we cannot use web browsers that require the graphical interface (like `firefox` or `chrome`). Instead, we are going to use `lynx`, a text-based web browser. To see the web page, you can go to **host2** and execute the command:

```
host2$ lynx localhost
```

As you can see, the web page is very simple, it only contains some few sentences and a link towards another local web page. You can browse that other web page by using the arrow keys. Obviously, we do not pretend to connect to the web server locally, but remotelly from **host3**. So, let us start the experiment:

1. Delete the kernel routing cache executing the label `flushcache`.
2. Put all wiresharks listening traffic in all SimNet interfaces.
3. Connect to the web server of **host2** using `lynx` from **host3**:

   ```
   host3$ lynx 192.168.0.2
   ```

   (a) Can you see the main web page of **host2**?

   (b) Can you see any error message in the wiresharks?

   As you can see there is no problem to see the main web page of this host. However, browse the web in order to see the other local web page:

   (a) Did you noticed any error?

   (b) What do you see in the wiresharks?

   (c) Why do you think that the main web page is working, but the second one cannot be downloaded?

   (d) Try to solve the problem by changing the MSS in **R2** using the `MANGLE` table of `iptables`. Remember that you have to change the MSS in the proper message of the 3-way handshake.

We hope that all these experiments helped you to understand how complicated is to solve problems of communication networks when using tunnels, even in the case of using the most simplistic IPIP tunnel. In following lab sessions, maybe you will be asked to create GRE tunnels, so try to remember all these concepts.