

CS4150 Computer Networks Laboratory – Assignment 9

Subhash S – 111801042
Computer Science and Engineering, IIT Palakkad

MTU of r1

Maximum Transmission Unit (MTU) is the size of the largest protocol data unit (PDU) that can be communicated in a single network layer transaction.

Figure 1: With data size of 383 and DF flag set, there are no issues.

```
Ubuntu-20.04
tc@h1:~$ sudo hping3 -V --dontfrag -d 383 192.168.101.2
using eth1, addr: 192.168.1.2, MTU: 1500
HPING 192.168.101.2 (eth1 192.168.101.2): NO FLAGS are set, 40 headers + 383 data bytes
```

Figure 2: With data size of 384 and the DF flag set, the network layer raises an issue.

```
Ubuntu-20.04
tc@h1:~$ sudo hping3 -V --dontfrag -d 384 192.168.101.2
using eth1, addr: 192.168.1.2, MTU: 1500
HPING 192.168.101.2 (eth1 192.168.101.2): NO FLAGS are set, 40 headers + 384 data bytes
ICMP Fragmentation Needed/DF set from ip=192.168.1.1 get hostname ...
```

When I **set** the **don't fragment** flag, there are no issues in sending a data size of 383B but there are issues when sending a data size of 384B which means that the networking layer has an MTU of **423B**.

Note: When I **unset** the **don't fragment** flag, packet size turns out to be **420B**.

Figure 3: The size of the packet along with the ethernet frame header (underlined in red)

```
tc@r2:~$ sudo tcpdump -i eth2 -e src 192.168.1.2 and tcp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth2, link-type EN10MB (Ethernet), capture size 65535 bytes
03:55:10.032084 08:00:27:d0:7c:cd (oui Unknown) > 08:00:27:a6:ef:5d (oui Unknown), ethertype IPv4 (0x0800), length 434: 192.168.1.2.1108 > 192.168.101.2: tcp
Flags [], seq 816616158:816616538, win 512, length 380
03:55:10.032096 08:00:27:d0:7c:cd (oui Unknown) > 08:00:27:a6:ef:5d (oui Unknown), ethertype IPv4 (0x0800), length 434: 192.168.1.2 > 192.168.101.2: tcp
03:55:10.032097 08:00:27:d0:7c:cd (oui Unknown) > 08:00:27:a6:ef:5d (oui Unknown), ethertype IPv4 (0x0800), length 434: 192.168.1.2 > 192.168.101.2: tcp
03:55:10.032099 08:00:27:d0:7c:cd (oui Unknown) > 08:00:27:a6:ef:5d (oui Unknown), ethertype IPv4 (0x0800), length 60: 192.168.1.2 > 192.168.101.2: tcp
03:55:11.032218 08:00:27:d0:7c:cd (oui Unknown) > 08:00:27:a6:ef:5d (oui Unknown), ethertype IPv4 (0x0800), length 434: 192.168.1.2.kpop > 192.168.101.2: tcp
Flags [], seq 118206968:118207348, win 512, length 380
03:55:11.032224 08:00:27:d0:7c:cd (oui Unknown) > 08:00:27:a6:ef:5d (oui Unknown), ethertype IPv4 (0x0800), length 434: 192.168.1.2 > 192.168.101.2: tcp
03:55:11.032226 08:00:27:d0:7c:cd (oui Unknown) > 08:00:27:a6:ef:5d (oui Unknown), ethertype IPv4 (0x0800), length 434: 192.168.1.2 > 192.168.101.2: tcp
03:55:11.032443 08:00:27:d0:7c:cd (oui Unknown) > 08:00:27:a6:ef:5d (oui Unknown), ethertype IPv4 (0x0800), length 60: 192.168.1.2 > 192.168.101.2: tcp
03:55:12.032592 08:00:27:d0:7c:cd (oui Unknown) > 08:00:27:a6:ef:5d (oui Unknown), ethertype IPv4 (0x0800), length 434: 192.168.1.2.1110 > 192.168.101.2: tcp
Flags [], seq 142569198:142569578, win 512, length 380
```

When I **unset** the **don't fragment** flag, the above figure shows the size of the packet along with the link layer headers and the total length here is **434B** (420B obtained above + 14B ethernet headers).

When I **set** the **don't fragment** flag, the size turns out to be **437B** (423B above + 14B ethernet headers).

So the MTU of r1 is **423B**, this can also be confirmed from the **ifconfig** command in r1. But the value of x is **434B** since the fragmentation bit was not set and the ethernet frame headers were taken into MTU calculation.

Hence the value of x is **434**.

Downloading packets from r1

```
/mnt/i/Networks-Lab/Lab9
→ scp -P 14601 tc@localhost:ipfrags.tar.xz .
tc@localhost's password:
ipfrags.tar.xz      100% 13KB 4.5MB/s 00:00
```

Filtering out invalid fragments

File: q2.cpp, Run the “make” command to generate the binary.

Aim

To validate the checksum and filter out invalid fragments from the list of 540 give fragments.

Functions and Classes

bitset: This is a class in C++ than be used to work with binary strings of arbitrary lengths.

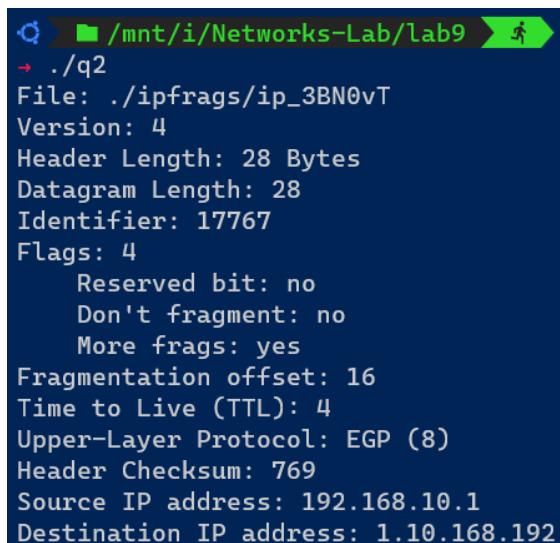
add: This function adds two 16-bit binary strings, if there is a carry, it adds the carry back to the result and returns the final result.

Program

This program takes an optional argument which specifies the directory in which the ip fragments are located. This directory has to contain only these 540 files and nothing else.

1. Using *dirent*, I go over each file in the directory and store its name in a vector of strings named *file_names*.
2. For each file in *file_names*:
 - a. Using *ifstream*, I read the file in binary mode into the packet structure.
 - b. Now I cast the packet structure into a *uint16_t* pointer as this will be helpful in obtaining 2 bytes of the headers at a time and incrementing the pointer would increment it by 2 bytes.
 - c. I sum the current 16 bits part of the header to the result (initialised to 0) using the *add* function described above.
 - d. I increment *uint16_t* pointer to move to the 16 bits of the header and continue from c.
 - e. If all the bits in the final sum (obtained after summing up all the parts of the headers) are set, then the checksum is valid and I print the header details of this fragment and write the filename to a file named *valid_fragment_files* which will be used by *q3.cpp* to assemble and find the data contained in the packets.

Figure 4: A part of the output from q2. Prints all the header details along with flags.



```
→ ./q2
File: ./ipfrags/ip_3BN0vT
Version: 4
Header Length: 28 Bytes
Datagram Length: 28
Identifier: 17767
Flags: 4
    Reserved bit: no
    Don't fragment: no
    More frags: yes
Fragmentation offset: 16
Time to Live (TTL): 4
Upper-Layer Protocol: EGP (8)
Header Checksum: 769
Source IP address: 192.168.10.1
Destination IP address: 1.10.168.192
```

Assembling all the fragments into packets to find the data

File: q3.cpp, Run the “make” command to generate the binary.

Aim

To read the files with valid fragments and assemble them into one or more packets depending on the identifiers in the fragment headers.

Functions and Classes

CompareFrag: This class has a publicly overloaded operator method that compares two fragments. The comparison between two *IPPkets* a & b is as follows:

1. If either a or b have their *more flags* header bit unset, then they appear at the end of list hence they are given the least priority.
2. If the *fragment offset* is the same, then we can discard packets with no data.
3. If both the above cases don't pass then the packet with a smaller *fragment offset* is given a higher priority since the *fragment offsets* increase for contiguous fragments in the same packet.

This comparison is used by the priority queue which is used to obtain fragments in an order.

Program

I am using a map between the id and a priority queue of fragments (with comparator as mentioned above) since we know that all the fragments with the same id belong to the same packet.

1. Since q2 writes all the valid fragment's path to the file named *valid_fragment_files*, I read this file and store the list of all valid fragments in a vector.
2. For each file obtained in 1, I read the file in binary mode using *ifstream* and store it in the packet structure. This packet structure is then added to appropriate priority queue depending on the packet id.
3. For each of the key, value pair in the map,
 - a. Since the fragments are stored in the priority queue with a custom comparator, each time I peek for a fragment from the queue, I will them in the required order. After popping each packet, I read *total length – header length* amount of bytes from the *data* field of the structure and append it to the result and using the *fragment offset* field I can decide where exactly in the result buffer I need to append this data.
 - b. I keep track of the total number of fragments in this packet through the *num_frgs* variable.
 - c. I pop the fragment once it has been used and go back to a.
 - d. After the priority queue is empty, I print the number of fragments, id of the packet, total length of packet and the data contained in the result buffer.

A total of 3 packets were obtained and their sizes are as follows:

1. Packet with ID **17767** has a total size of **220B** (192B data + 28B headers)
2. Packet with ID **20026** has a total size of **223B** (195B data + 28B headers)
3. Packet with ID **38906** has a total size of **220B** (192B data + 28B headers)

Figure 5: The output after running q3. It shows the packet size and the data contained in the packet.

```
🔗 /mnt/i/Networks-Lab/lab9 ➤  
➤ ./q3  
  
Total number of IPv4 packets: 3  
  
Packet in 19 fragments  
ID: 17767  
Packet size: 220 Bytes  
Data size: 192 Bytes  
Data: The checksum field is the 16-bit one's complement of the one's complement sum of all 16-bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero  
  
Packet in 17 fragments  
ID: 20026  
Packet size: 223 Bytes  
Data size: 195 Bytes  
Data: rand() function is used in C to generate random numbers. If we generate a sequence of random number with rand() function, it will create the same sequence again and again every time program runs.  
  
Packet in 18 fragments  
ID: 38906  
Packet size: 220 Bytes  
Data size: 192 Bytes  
Data: Since the 1980s, it was apparent that the pool of available IPv4 addresses was being depleted at a rate that was not initially anticipated in the original design of the network address system.
```