Etapa 5 - Integração

Grupo:
Guilherme Nishi Kanashiro - 628298
Leonardo Utida Alcântara - 628182
Rodolfo Krambeck Asbahr - 628042
Tiago Bachiega de Almeida - 628247

Como executar o programa:

Primeiramente, é necessário ajustar o mtu dele, pois o tamanho padrão é de 65536 bytes, o que impossibilita qualquer forma de fragmentação automática e queremos testar o que foi feito na Etapa 3 integrada com o código da Etapa 4. Para fazer esse ajuste, é necessário usar o seguinte comando:

```
$ ip link set lo mtu 1500
```

Feito isso, é necessário ajustar dentro do código os seguintes parâmetros de acordo com sua rede e dispositivo. Para o grupo, foi utilizado o seguinte:

```
# Coloque aqui o endereço de destino para onde você quer mandar o ping
# Pode ser obtido com o comando arp -a | grep _gateway
dest_ip = '192.168.1.1'

# Coloque abaixo o endereço IP do seu computador na sua rede local
# Pode ser obtido com o comando ifconfig
src_ip = '192.168.1.116'

# Coloque aqui o nome da sua placa de rede
# Pode ser obtido com o comando ifconfig
if_name = 'wlp2s0'

# Coloque aqui o endereço MAC do roteador da sua rede local (arp -a | grep _gateway)
# Pode ser obtido com o comando arp -a | grep _gateway
dest_mac = '7c:8b:ca:3d:01:8e'

# Coloque aqui o endereço MAC da sua placa de rede (ip link show dev wlan0)
# Pode ser obtido com  ip link show dev <nomePlacaDeRede>
src_mac = '28:e3:47:b0:6f:fa'
```

A implementação da interpretação e reconstrução dos datagramas IPs foram feitas utilizando python 3. Para rodar o código em um terminal é preciso executar o seguinte comando:

```
$ sudo python3 etapa5.py
```

Como funciona o programa:

Adição do tratamento da camada de transporte

Na Etapa 4 tínhamos um código que unia o tratamento da Camada de Rede e da Camada de Enlace. O objetivo agora foi adicionar também o tratamento da Camada de Transporte. Para isso, foram feitas algumas adições e modificações com base no que foi feito na Etapa 2.

De início, foram adicionadas no código da Etapa 4 todas as funções, variáveis, classes e parâmetros da Etapa 2. A função que na Etapa 2 implementava a Máquina de Estados da Camada de Transporte agora aparece no programa como a função *transportLayer()*. Esta função recebe um pacote e o trata de maneira idêntica ao que foi feito na Etapa 2.

As funções da Etapa 2 *enviaSrc()* e *enviaDst()* foram modificadas para o formato a seguir:

```
def enviaSrc(fd, conexao, segment):
    (src_addr, src_port, dst_addr, dst_port) = conexao.id_conexao
    send_ip(fd, segment, ICMP)

def enviaDst(fd, conexao, segment):
    (src_addr, src_port, dst_addr, dst_port) = conexao.id_conexao
    send_ip(fd, segment, ICMP)
```

Estas funções são chamadas em trechos do código como o que se segue:,

```
segment = struct.pack('!HHIIHHHH', src_port, dst_port, conexao.seq_no,
                      conexao.ack_no, (5<<12)|FLAGS_ACK,
                      1024, 0, 0) + payload

conexao.seq_no = (conexao.seq_no + len(payload)) & 0xffffffff

enviaDst(fd, conexao, segment)
```

A adição da Camada de Transporte no processamento se dá no *raw_recv()*, logo depois da fragmentação da mensagem, que foi modificada para que retornasse algo na variável *defragMsg* apenas se fosse uma mensagem completa, pronta para ir para a Camada de Transporte. Caso isso ocorra, o código envia a mensagem para a função *transportLayer()*. Ali é feito o tratamento conforme foi implementado na Etapa 2.

```python
def raw_recv(fd):
    frame = fd.recv(12000)

    print("\n ========================== \n")

    #Verifica MAC e Protocol
    if verify_MAC(frame) and verify_protocol(frame):
        print('recebido quadro de %d bytes' % len(frame))
        print(repr(frame))

        #Obtem o Datagram
        datagram = frame[14:]

        #Obtem a mensagem fragmentada
        defragMSG = defrag(datagram)

        #Isso impede que o um fragmento de mensagem va para o
        #transportLayer
        if defragMSG is not None:
            #vai para a camada de transporte
            transportLayer(defragMSG)

    else:
        print("Nao passou na verificacao")
```

Teste:

Para testar o código, deixamos o programa rodando por algum tempo e verificamos quais mensagens de retorno ele mostrava no terminal.

A imagem a seguir é um exemplo de tratamento correto. É possível verificar que o quadro passou na verificação de MAC Address e de protocolo. com isso, mostra-se a mensagem completa de 66 bytes, que não se fragmentou. Com isso, o pacote é enviado à Camada de Transporte. Pelas mensagens exibidas, pode-se verificar o correto funcionamento do tratamento, indo desde a confirmação do ACK até a informação sobre em qual dos estados da Máquina de Estados o programa se encontra, sendo Slow Start no caso.

```
MAC de destino :  28:e3:47:b0:6f:fa
Meu MAC :  28:e3:47:b0:6f:fa

****Verficando o protocolo****

Protocolo do Frame :  b'\x08\x00'
Protocolo IP:   b'\x08\x00'
recebido quadro de 66 bytes
b'(\xe3G\xb0o\xfa|\x8b\xca=\x01\x8e\x08\x00E\x00\x004\xeb\x1e@\x006\x06n(\xb9\xc7o\x9
9\xc0\xa8\x01t\x01\xbb\xec6\xbcO\xc7\xf5\xddJ\x88U\x80\x10\x00=\x83\x9b\x00\x00\x01\x
01\x08\n\x91!\x8aZ\x8a_\x89\xb4'

****Transport Layer****

Entrou em ack_recv
ack_no = 3712649301
con.send_base = 3712649221
Entrou no ack >
Confirmando ack:
 3712649301
Dados reconhecidos:
 80
Estado: Slow Start
```

Na imagem a seguir, é possível se ter um exemplo de caso onde não foi verificado o MAC Address correto, então o quadro foi ignorado, como era o esperado.

Também é possível verificar o caso onde a Camada de Transporte detecta uma conexão desconhecida e não processa o pacote, como na imagem seguinte.

Por fim, temos um caso onde o programa se comporta de forma bastante estranha. O grupo não conseguiu descobrir a causa do problema a seguir, onde fica em um loop bem grande enviando o quadro atual e processando o próximo envio. Eventualmente o programa sai deste loop e volta à sua execução normal.