



# Tracert desarrollado con Python

Nombre: Eduardo Eliezar Castillo Hernández





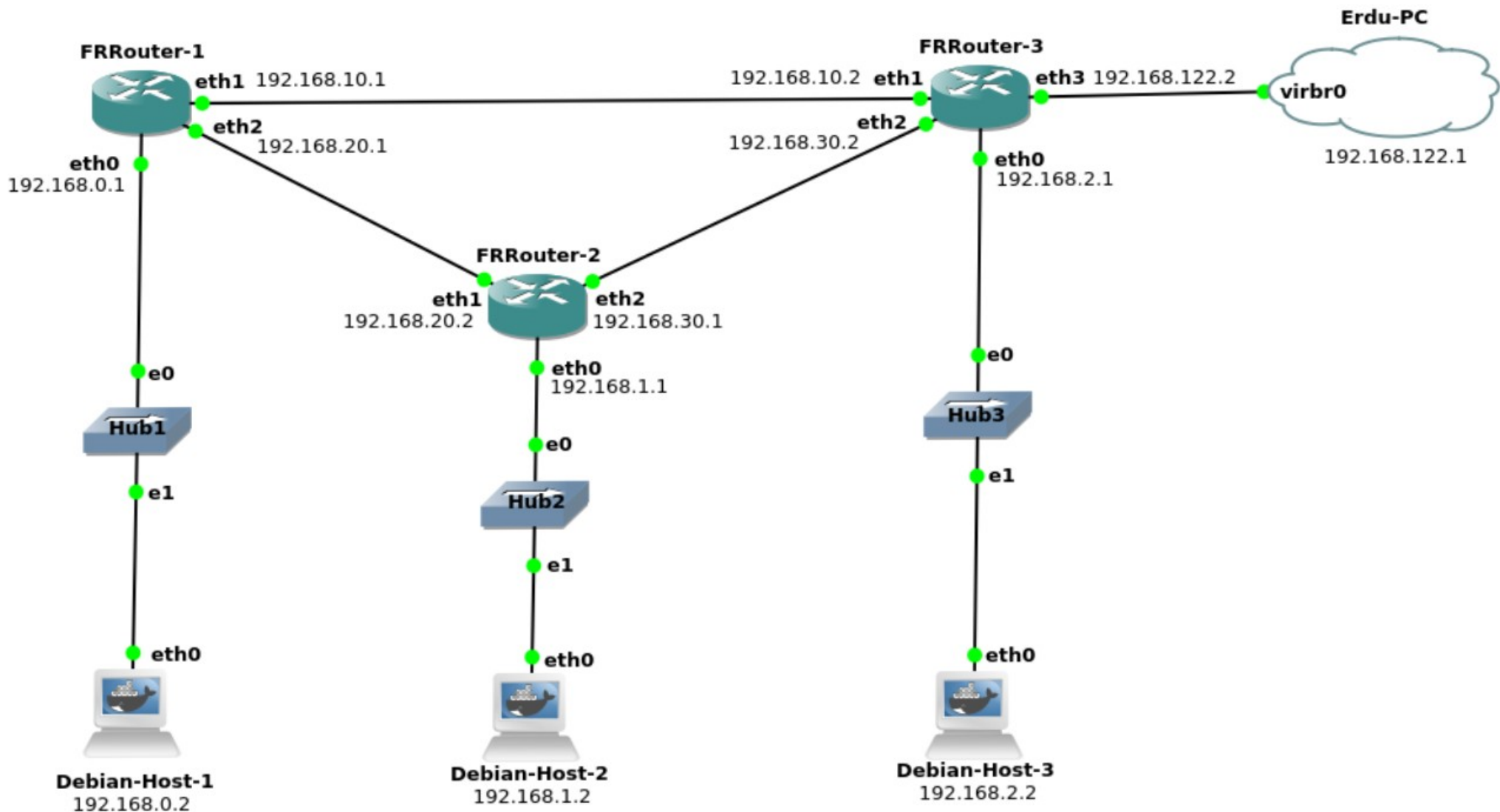
# PseudoCodigo

```
UDP = crear_socketUDP();
RAW = crear_socketRAW();
RAW → establecer_Tiempo_de_Espera(5);
RAW → vincular_con(IP, Puerto);
Mientras (true){
    Tiempo_inicio = tiempoActual();
    UDP → enviar_Datagrama(IP_Destino, Puerto_destino, TTL);
    Paquete = RAW → recibir_ICMP();
    Tiempo_final = tiempoActual();
    RTT = (Tiempo_final – Tiempo_inicio) * 1000;
    imprimir_Informe(IP_Recibida, Puerto, RTT, Paquete);
    Si (Paquete → Codigo_ICMP = 3   Y   Paquete → Tipo_ICMP = 3)
        Terminar_ciclo;

    TTL = TTL + 1;
}
```

# Topologia

Simulación del código al trazar la ruta de “Erdu-PC” a “Debian-Host-1”





# Python3 tracer.py -a 192.168.0.2

```
#Extrayendo parametros
HOST = args['<destino>'] #IP Destino ó host de destino
PORT = int(args['-p']) #Puerto destino
MAX_HOPS = int(args['-s']) #Nº Saltos maximos permitidos
ADVANCED_MODE = args["-a"] #Activacion del modo detallado

#Obteniendo una direccion IP si el host de destino fue especificado
#mediante un nombre (Ej: www.google.com)
IP_ADDRESS = (socket.gethostbyname(HOST), int(PORT))

#Creando RAW socket
raw = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

#Estableciendo el tiempo que se debe esperar para recibir
#un paquete, mediante la siguiente estructura:
#
# struct timeval{
#     long segundos;
#     long milisegundos;
# }
# TimeOut = 5 Seg
raw.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack("ll", 5, 0))

#Vinculando raw socket a la interfaz activa en el puerto especificado
raw.bind(('', PORT))

#Creando socket UDP
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

#Imprimiendo los datos pasados por la línea de comandos.
print("Trazando ruta hacia {} ({} ) al puerto {} con {} saltos maximos\n".format(
    HOST, IP_ADDRESS[0], PORT, MAX_HOPS
))
```

**VARIABLES GLOBALES:**  
HOST =



# Python3 tracer.py -a 192.168.0.2

```
#Extrayendo parametros
HOST = args['<destino>'] #IP Destino ó host de destino
PORT = int(args['-p']) #Puerto destino
MAX_HOPS = int(args['-s']) #Nº Saltos maximos permitidos
ADVANCED_MODE = args["-a"] #Activacion del modo detallado

#Obteniendo una direccion IP si el host de destino fue especificado
#mediante un nombre (Ej: www.google.com)
IP_ADDRESS = (socket.gethostbyname(HOST), int(PORT))

#Creando RAW socket
raw = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

#Estableciendo el tiempo que se debe esperar para recibir
#un paquete, mediante la siguiente estructura:
#
# struct timeval{
#     long segundos;
#     long milisegundos;
# }
# TimeOut = 5 Seg
raw.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack("ll", 5, 0))

#Vinculando raw socket a la interfaz activa en el puerto especificado
raw.bind(('', PORT))

#Creando socket UDP
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

#Imprimiendo los datos pasados por la línea de comandos.
print("Trazando ruta hacia {} ({} ) al puerto {} con {} saltos maximos\n".format(
    HOST, IP_ADDRESS[0], PORT, MAX_HOPS
))
```

## VARIABLES GLOBALES:

HOST = 192.168.0.2

PORT =



# Python3 tracer.py -a 192.168.0.2

```
#Extrayendo parametros
HOST = args['<destino>'] #IP Destino ó host de destino
PORT = int(args['-p']) #Puerto destino
MAX_HOPS = int(args['-s']) #Nº Saltos maximos permitidos
ADVANCED_MODE = args["-a"] #Activacion del modo detallado

#Obteniendo una direccion IP si el host de destino fue especificado
#mediante un nombre (Ej: www.google.com)
IP_ADDRESS = (socket.gethostbyname(HOST), int(PORT))

#Creando RAW socket
raw = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

#Estableciendo el tiempo que se debe esperar para recibir
#un paquete, mediante la siguiente estructura:
#
# struct timeval{
#     long segundos;
#     long milisegundos;
# }
# Timeout = 5 Seg
raw.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack("ll", 5, 0))

#Vinculando raw socket a la interfaz activa en el puerto especificado
raw.bind(('', PORT))

#Creando socket UDP
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

#Imprimiendo los datos pasados por la línea de comandos.
print("Trazando ruta hacia {} ({{}}) al puerto {} con {} saltos maximos\n".format(
    HOST, IP_ADDRESS[0], PORT, MAX_HOPS
))
```

## VARIABLES GLOBALES:

HOST = 192.168.0.2

PORT = 33434

MAX\_HOPS =



# Python3 tracer.py -a 192.168.0.2

```
#Extrayendo parametros
HOST = args['<destino>'] #IP Destino ó host de destino
PORT = int(args['-p']) #Puerto destino
MAX_HOPS = int(args['-s']) #Nº Saltos maximos permitidos
ADVANCED_MODE = args["-a"] #Activacion del modo detallado

#Obteniendo una direccion IP si el host de destino fue especificado
#mediante un nombre (Ej: www.google.com)
IP_ADDRESS = (socket.gethostbyname(HOST), int(PORT))

#Creando RAW socket
raw = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

#Estableciendo el tiempo que se debe esperar para recibir
#un paquete, mediante la siguiente estructura:
#
# struct timeval{
#     long segundos;
#     long milisegundos;
# }
# Timeout = 5 Seg
raw.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack("ll", 5, 0))

#Vinculando raw socket a la interfaz activa en el puerto especificado
raw.bind(('', PORT))

#Creando socket UDP
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

#Imprimiendo los datos pasados por la línea de comandos.
print("Trazando ruta hacia {} ({{}}) al puerto {} con {} saltos maximos\n".format(
    HOST, IP_ADDRESS[0], PORT, MAX_HOPS
))
```

## VARIABLES GLOBALES:

HOST = 192.168.0.2

PORT = 33434

MAX\_HOPS = 30

ADVANCED\_MODE =



# Python3 tracer.py -a 192.168.0.2

```
#Extrayendo parametros
HOST = args['<destino>'] #IP Destino ó host de destino
PORT = int(args['-p']) #Puerto destino
MAX_HOPS = int(args['-s']) #Nº Saltos maximos permitidos
ADVANCED_MODE = args["-a"] #Activacion del modo detallado

#Obteniendo una direccion IP si el host de destino fue especificado
#mediante un nombre (Ej: www.google.com)
IP_ADDRESS = (socket.gethostbyname(HOST), int(PORT))

#Creando RAW socket
raw = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

#Estableciendo el tiempo que se debe esperar para recibir
#un paquete, mediante la siguiente estructura:
#
# struct timeval{
#     long segundos;
#     long milisegundos;
# }
# TimeOut = 5 Seg
raw.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack("ll", 5, 0))

#Vinculando raw socket a la interfaz activa en el puerto especificado
raw.bind(('', PORT))

#Creando socket UDP
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

#Imprimiendo los datos pasados por la línea de comandos.
print("Trazando ruta hacia {} ({} ) al puerto {} con {} saltos maximos\n".format(
    HOST, IP_ADDRESS[0], PORT, MAX_HOPS
))
```

## VARIABLES GLOBALES:

HOST = 192.168.0.2

PORT = 33434

MAX\_HOPS = 30

ADVANCED\_MODE = *True*

IP\_ADDRESS =



# Python3 tracert.py -a 192.168.0.2

```
#Extrayendo parametros
HOST = args['<destino>'] #IP Destino ó host de destino
PORT = int(args['-p']) #Puerto destino
MAX_HOPS = int(args['-s']) #Nº Saltos maximos permitidos
ADVANCED_MODE = args["-a"] #Activacion del modo detallado

#Obteniendo una direccion IP si el host de destino fue especificado
#mediante un nombre (Ej: www.google.com)
IP_ADDRESS = (socket.gethostbyname(HOST), int(PORT))

#Creando RAW socket
raw = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

#Estableciendo el tiempo que se debe esperar para recibir
#un paquete, mediante la siguiente estructura:
#
# struct timeval{
#     long segundos;
#     long milisegundos;
# }
# TimeOut = 5 Seg
raw.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack("ll", 5, 0))

#Vinculando raw socket a la interfaz activa en el puerto especificado
raw.bind(('', PORT))

#Creando socket UDP
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

#Imprimiendo los datos pasados por la línea de comandos.
print("Trazando ruta hacia {} ({{}}) al puerto {} con {} saltos maximos\n".format(
    HOST, IP_ADDRESS[0], PORT, MAX_HOPS
))
```

## VARIABLES GLOBALES:

HOST = 192.168.0.2

PORT = 33434

MAX\_HOPS = 30

ADVANCED\_MODE = *True*

IP\_ADDRESS =  
(192.168.0.2, 33434)



# Python3 tracer.py -a 192.168.0.2

```
#Extrayendo parametros
HOST = args['<destino>'] #IP Destino ó host de destino
PORT = int(args['-p']) #Puerto destino
MAX_HOPS = int(args['-s']) #Nº Saltos maximos permitidos
ADVANCED_MODE = args["-a"] #Activacion del modo detallado

#Obteniendo una direccion IP si el host de destino fue especificado
#mediante un nombre (Ej: www.google.com)
IP_ADDRESS = (socket.gethostbyname(HOST), int(PORT))

#Creando RAW socket
raw = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

#Estableciendo el tiempo que se debe esperar para recibir
#un paquete, mediante la siguiente estructura:
#
# struct timeval{
#     long segundos;
#     long milisegundos;
# }
# Timeout = 5 Seg
raw.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack("ll", 5, 0))

#Vinculando raw socket a la interfaz activa en el puerto especificado
raw.bind(('', PORT))

#Creando socket UDP
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

#Imprimiendo los datos pasados por la linea de comandos.
print("Trazando ruta hacia {} ({} ) al puerto {} con {} saltos maximos\n".format(
    HOST, IP_ADDRESS[0], PORT, MAX_HOPS
))
```

## VARIABLES GLOBALES:

HOST = 192.168.0.2

PORT = 33434

MAX\_HOPS = 30

ADVANCED\_MODE = True

IP\_ADDRESS =  
(192.168.0.2, 33434)

## SOCKETS:

Socket UDP	Familia	Tipo	Protocolo
	Puerto	TTL	

Socket RAW	Familia	Tipo	Protocolo	IP	Puerto	Time Out



# Python3 tracer.py -a 192.168.0.2

```
#Extrayendo parametros
HOST = args['<destino>'] #IP Destino ó host de destino
PORT = int(args['-p']) #Puerto destino
MAX_HOPS = int(args['-s']) #Nº Saltos maximos permitidos
ADVANCED_MODE = args["-a"] #Activacion del modo detallado

#Obteniendo una direccion IP si el host de destino fue especificado
#mediante un nombre (Ej: www.google.com)
IP_ADDRESS = (socket.gethostbyname(HOST), int(PORT))

#Creando RAW socket
raw = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

#Estableciendo el tiempo que se debe esperar para recibir
#un paquete, mediante la siguiente estructura:
#
# struct timeval{
#     long segundos;
#     long milisegundos;
# }
# Timeout = 5 Seg
raw.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack("ll", 5, 0))

#Vinculando raw socket a la interfaz activa en el puerto especificado
raw.bind(('', PORT))

#Creando socket UDP
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

#Imprimiendo los datos pasados por la línea de comandos.
print("Trazando ruta hacia {} ({} ) al puerto {} con {} saltos maximos\n".format(
    HOST, IP_ADDRESS[0], PORT, MAX_HOPS
))
```

## VARIABLES GLOBALES:

HOST = 192.168.0.2

PORT = 33434

MAX\_HOPS = 30

ADVANCED\_MODE = True

IP\_ADDRESS =  
(192.168.0.2, 33434)

## SOCKETS:

Socket UDP	Familia	Tipo	Protocolo
	Puerto	TTL	

Socket RAW	Familia	Tipo	Protocolo	IP	Puerto	Time Out
	IPv4	RAW	ICMP			



# Python3 tracer.py -a 192.168.0.2

```
#Extrayendo parametros
HOST = args['<destino>'] #IP Destino ó host de destino
PORT = int(args['-p']) #Puerto destino
MAX_HOPS = int(args['-s']) #Nº Saltos maximos permitidos
ADVANCED_MODE = args["-a"] #Activacion del modo detallado

#Obteniendo una direccion IP si el host de destino fue especificado
#mediante un nombre (Ej: www.google.com)
IP_ADDRESS = (socket.gethostbyname(HOST), int(PORT))

#Creando RAW socket
raw = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

#Estableciendo el tiempo que se debe esperar para recibir
#un paquete, mediante la siguiente estructura:
#
# struct timeval{
#     long segundos;
#     long milisegundos;
# }
# Timeout = 5 Seg
raw.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack("ll", 5, 0))

#Vinculando raw socket a la interfaz activa en el puerto especificado
raw.bind(('', PORT))

#Creando socket UDP
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

#Imprimiendo los datos pasados por la línea de comandos.
print("Trazando ruta hacia {} ({{}}) al puerto {} con {} saltos maximos\n".format(
    HOST, IP_ADDRESS[0], PORT, MAX_HOPS
))
```

## VARIABLES GLOBALES:

HOST = 192.168.0.2

PORT = 33434

MAX\_HOPS = 30

ADVANCED\_MODE = True

IP\_ADDRESS =  
(192.168.0.2, 33434)

## SOCKETS:

Socket UDP	Familia	Tipo	Protocolo
	Puerto	TTL	

Socket RAW	Familia	Tipo	Protocolo	IP	Puerto	Time Out
	IPv4	RAW	ICMP			5 seg



# Python3 tracer.py -a 192.168.0.2

```
#Extrayendo parametros
HOST = args['<destino>'] #IP Destino ó host de destino
PORT = int(args['-p']) #Puerto destino
MAX_HOPS = int(args['-s']) #Nº Saltos maximos permitidos
ADVANCED_MODE = args["-a"] #Activacion del modo detallado

#Obteniendo una direccion IP si el host de destino fue especificado
#mediante un nombre (Ej: www.google.com)
IP_ADDRESS = (socket.gethostbyname(HOST), int(PORT))

#Creando RAW socket
raw = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

#Estableciendo el tiempo que se debe esperar para recibir
#un paquete, mediante la siguiente estructura:
#
# struct timeval{
#     long segundos;
#     long milisegundos;
# }
# Timeout = 5 Seg
raw.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack("ll", 5, 0))

#Vinculando raw socket a la interfaz activa en el puerto especificado
raw.bind(('', PORT))

#Creando socket UDP
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

#Imprimiendo los datos pasados por la línea de comandos.
print("Trazando ruta hacia {} ({} ) al puerto {} con {} saltos maximos\n".format(
    HOST, IP_ADDRESS[0], PORT, MAX_HOPS
))
```

## VARIABLES GLOBALES:

HOST = 192.168.0.2

PORT = 33434

MAX\_HOPS = 30

ADVANCED\_MODE = True

IP\_ADDRESS =  
(192.168.0.2, 33434)

## SOCKETS:

Socket UDP	Familia	Tipo	Protocolo
	Puerto	TTL	

Socket RAW	Familia	Tipo	Protocolo	IP	Puerto	Time Out
	IPv4	RAW	ICMP	-	33434	5 seg



# Python3 tracer.py -a 192.168.0.2

```
#Extrayendo parametros
HOST = args['<destino>'] #IP Destino ó host de destino
PORT = int(args['-p']) #Puerto destino
MAX_HOPS = int(args['-s']) #Nº Saltos maximos permitidos
ADVANCED_MODE = args["-a"] #Activacion del modo detallado

#Obteniendo una direccion IP si el host de destino fue especificado
#mediante un nombre (Ej: www.google.com)
IP_ADDRESS = (socket.gethostbyname(HOST), int(PORT))

#Creando RAW socket
raw = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

#Estableciendo el tiempo que se debe esperar para recibir
#un paquete, mediante la siguiente estructura:
#
# struct timeval{
#     long segundos;
#     long milisegundos;
# }
# Timeout = 5 Seg
raw.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack("ll", 5, 0))

#Vinculando raw socket a la interfaz activa en el puerto especificado
raw.bind(('', PORT))

#Creando socket UDP
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)

#Imprimiendo los datos pasados por la línea de comandos.
print("Trazando ruta hacia {} ({} ) al puerto {} con {} saltos maximos\n".format(
    HOST, IP_ADDRESS[0], PORT, MAX_HOPS
))
```

## VARIABLES GLOBALES:

HOST = 192.168.0.2

PORT = 33434

MAX\_HOPS = 30

ADVANCED\_MODE = True

IP\_ADDRESS =  
(192.168.0.2, 33434)

## SOCKETS:

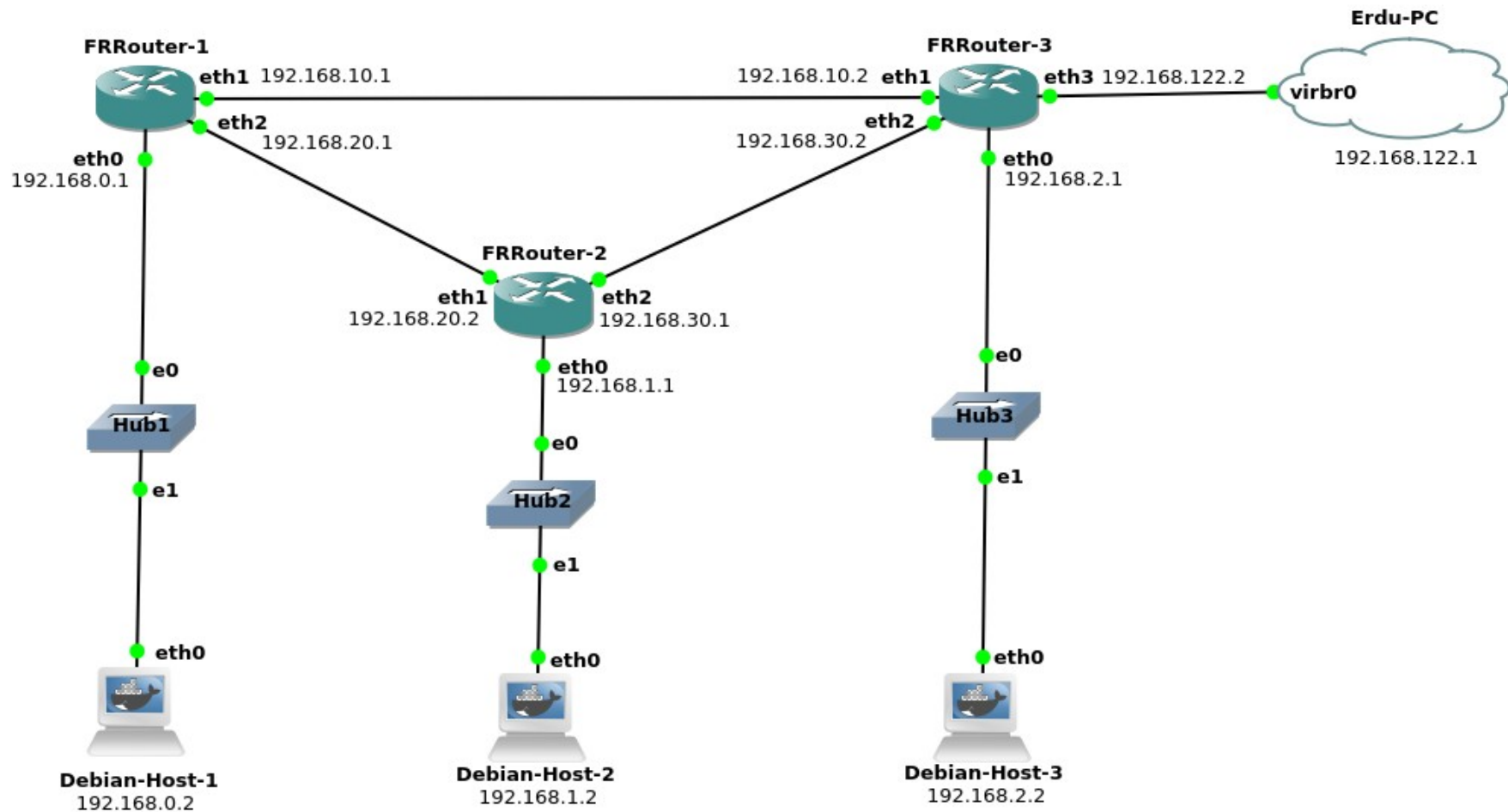
Socket UDP	Familia	Tipo	Protocolo
	IPv4	DGRAM	UDP
	Puerto	TTL	
	-		

Socket RAW	Familia	Tipo	Protocolo	IP	Puerto	Time Out
	IPv4	RAW	ICMP	-	33434	5 seg



# Python3 tracert.py -a 192.168.0.2

```
erdu@PC-Erdu:~/Documentos/Python/tracert$ sudo python3 tracert.py -a 192.168.0.2  
Trazando ruta hacia 192.168.0.2 (192.168.0.2) al puerto 33434 con 30 saltos maximos
```





# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE =  
ACT\_TTL =

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-		UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (0,0)  
ACT\_TTL =

## SOCKETS:

Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-		UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (0,0)  
ACT\_TTL = 0

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-		UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (0,0)  
ACT\_TTL = 0

## Listas:

Timer	packets	Packets_icmp	Packets_udp

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-		UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (0,0)  
ACT\_TTL = 0

## Listas:

Timer	packets	Packets_icmp	Packets_udp

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-		UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (0,0)  
ACT\_TTL = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-		UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (0,0)  
ACT\_TTL = 1  
i =

## Listas:

Timer	packets	Packets_icmp	Packets_udp

## SOCKETS:

Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (0,0)  
ACT\_TTL = 1  
i = 0

## Listas:

Timer	packets	Packets_icmp	Packets_udp

## SOCKETS:

Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (0,0)

ACT\_TTL = 1

i = 0

## Listas:

Timer	packets	Packets_icmp	Packets_udp
1.4519813			

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (0,0)

ACT\_TTL = 1

i = 0

## Listas:

Timer	packets	Packets_icmp	Packets_udp
1.4519813			

## SOCKETS:

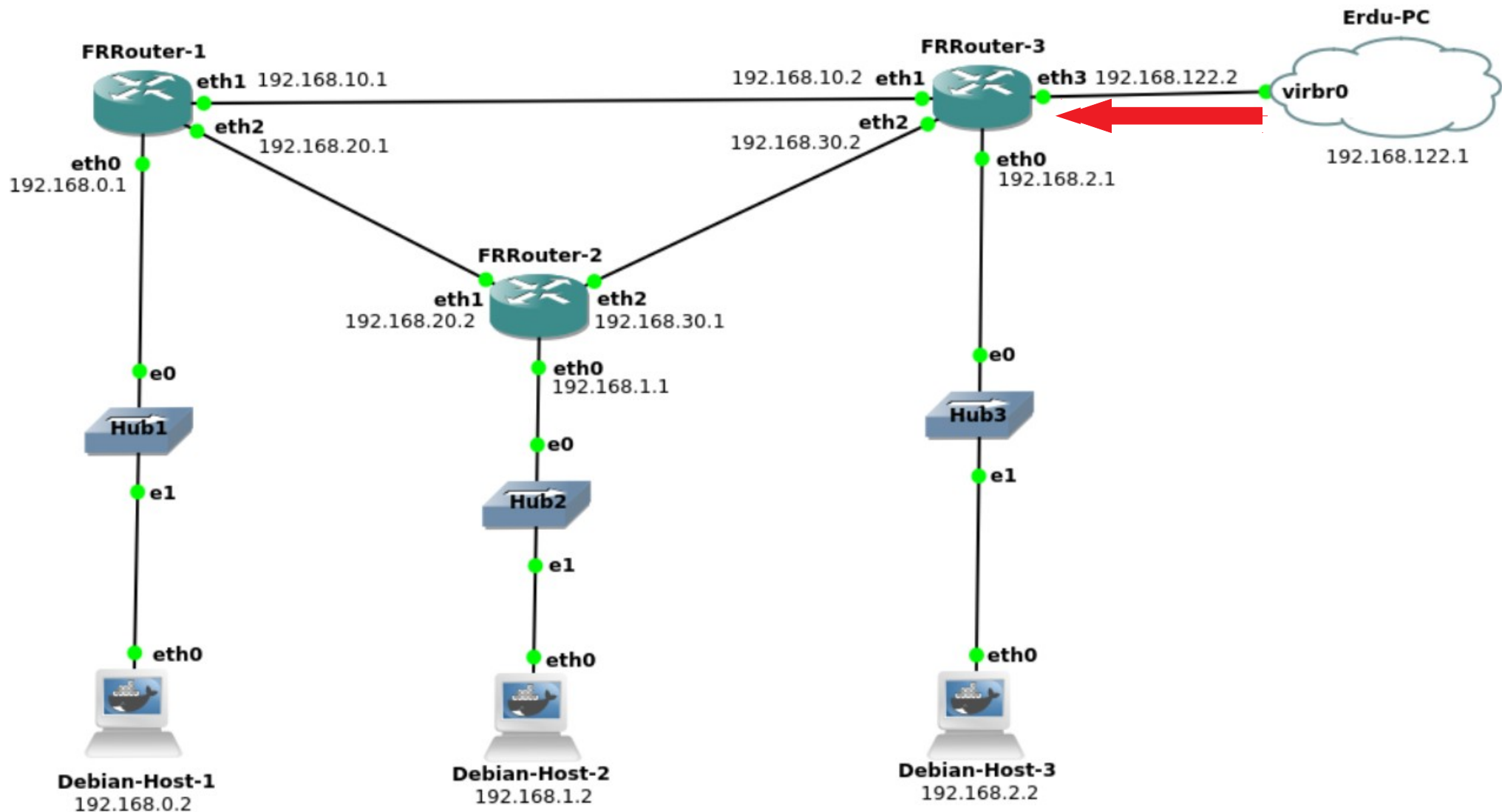
Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP



# Topologia

Simulación del código al trazar la ruta de “Erdu-PC” a “Debian-Host-1”





# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 0

## Listas:

Timer	packets	Packets_icmp	Packets_udp
1.4519813			

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 0

## Listas:

Timer	packets	Packets_icmp	Packets_udp
1.4519813			

$$\text{timer}[i] = (1.4525913 - 1.4519813) * 1000 = 0.61$$

## SOCKETS:

Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 0

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61			

## SOCKETS:

$$\text{timer}[i] = (1.4525913 - 1.4519813) * 1000 = 0.61$$

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 0

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
1.4526205			

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
1.4526205			

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
1.4526205			

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38			

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 2

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 2

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
1.4530149			

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 2

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
1.4530149			

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 2

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
1.4530149			

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
IP_RECEIVE = (0,0) #Variable que almacena la IP del ultimo ICMP recibido.
ACT_TTL = 0 #Variable que almacena el TTL con el que se trabaja en cada salto.

while IP_ADDRESS[0] != IP_RECEIVE[0]:
    timer = list() #Almacena los tiempos de envio y recepcion de cada paquete UDP.
    packets = list() #Almacena datagramas IP recibidos.
    packets_icmp = list() #Almacena los paquetes ICMP recibidos.
    packets_udp = list() #Almacena los datagramas UDP originales.

    ACT_TTL += 1 #Aumentando el ACT_TTL, para establecer el TTL actual.

    #Especificando el TTL que utilizara el socket UDP al enviar los datagramas.
    udp.setsockopt(socket.SOL_IP, socket.IP_TTL, ACT_TTL)

    #Iniciando un ciclo que se repetira 3 veces, una por cada datagrama UDP que se debe e
    for i in range(0, 3):
        timer.append(time.time()) #Guardando el tiempo en que se envia el datagrama UDP.

        #Enviando datagrama UDP a la direccion especificada por "IP_ADDRESS".
        udp.sendto("Tracert".encode(), IP_ADDRESS)

        try:
            #Recibiendo ICMP de error, con una longitud maxima de 1480 bytes.
            DATA, IP_RECEIVE = raw.recvfrom(1480)
        except BlockingIOError:
            #Si el tiempo de espera se agoto, en vez de almacenar el RTT se guarda "???".
            timer[i] = "???"
            continue
        else:
            #Si recibimos respuesta, calculamos el RTT en milisegundos
            timer[i] = round((time.time() - timer[i]) * 1000, 2)

            #Y guardamos los bytes (Paquetes IP) recibidos en la lista.
            packets.append(DATA)
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)  
ACT\_TTL = 1  
i = 2

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
0.32	n Bytes		

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
#Imprimiendo informe.
print("> TTL =", ACT_TTL, "\t", end='')

if (len(packets) == 0): #Si no recibimos ni un solo ICMP de error, entonces imprimimos "* * *".
    print["* * *\n"]
else:
    try:
        #Obteniendo el nombre asociado a la IP desde la que nos enviaron el paquete ICMP.
        IP_RECEIVE_NAME = socket.gethostbyaddr(IP_RECEIVE[0])[0]
    except socket.herror:
        #Si la IP no tiene asociada ningun paquete, entonces guardamos "-"
        IP_RECEIVE_NAME = '-'

    #Imprimiendo la IP y el nombre (si lo tiene) del que envio el ICMP, mas el RTT de cada respuesta.
    print("{} ({})\t\t{}\n".format(
        IP_RECEIVE[0], IP_RECEIVE_NAME, [str(n) + " ms" for n in timer]
    ))
```

## VARIABLES GLOBALES:

IP\_ADDRESS = (192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)

ACT\_TTL = 1

## SOCKETS:

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
0.32	n Bytes		

Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
#Imprimiendo informe.
print("> TTL =", ACT_TTL, "\t", end='')

if (len(packets) == 0): #Si no recibimos ni un solo ICMP de error, entonces imprimimos "* * *".
    print["* * *\n"]
else:
    try:
        #Obteniendo el nombre asociado a la IP desde la que nos enviaron el paquete ICMP.
        IP_RECEIVE_NAME = socket.gethostbyaddr(IP_RECEIVE[0])[0]
    except socket.herror:
        #Si la IP no tiene asociada ningun paquete, entonces guardamos "-"
        IP_RECEIVE_NAME = '-'

    #Imprimiendo la IP y el nombre (si lo tiene) del que envio el ICMP, mas el RTT de cada respuesta.
    print("{} ({})\t\t{}\n".format(
        IP_RECEIVE[0], IP_RECEIVE_NAME, [str(n) + " ms" for n in timer]
    ))
```

## VARIABLES GLOBALES:

IP\_ADDRESS = (192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)

ACT\_TTL = 1

## SOCKETS:

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
0.32	n Bytes		

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
#Imprimiendo informe.
print("> TTL =", ACT_TTL, "\t", end='')

if (len(packets) == 0): #Si no recibimos ni un solo ICMP de error, entonces imprimimos "* * *".
    print("\n")
else:
    try:
        #Obteniendo el nombre asociado a la IP desde la que nos enviaron el paquete ICMP.
        IP_RECEIVE_NAME = socket.gethostbyaddr(IP_RECEIVE[0])[0]
    except socket.herror:
        #Si la IP no tiene asociada ningun paquete, entonces guardamos "-"
        IP_RECEIVE_NAME = '-'

    #Imprimiendo la IP y el nombre (si lo tiene) del que envio el ICMP, mas el RTT de cada respuesta.
    print("{} ({})\t\t{}\n".format(
        IP_RECEIVE[0], IP_RECEIVE_NAME, [str(n) + " ms" for n in timer]
    ))
```

## VARIABLES GLOBALES:

IP\_ADDRESS = (192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)

IP\_RECEIVE\_NAME =

ACT\_TTL = 1

## SOCKETS:

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
0.32	n Bytes		

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
#Imprimiendo informe.
print("> TTL =", ACT_TTL, "\t", end='')

if (len(packets) == 0): #Si no recibimos ni un solo ICMP de error, entonces imprimimos "* * *".
    print["* * *\n"]
else:
    try:
        #Obteniendo el nombre asociado a la IP desde la que nos enviaron el paquete ICMP.
        IP_RECEIVE_NAME = socket.gethostbyaddr(IP_RECEIVE[0])[0]
    except socket.herror:
        #Si la IP no tiene asociada ningun paquete, entonces guardamos "-"
        IP_RECEIVE_NAME = '-'

    #Imprimiendo la IP y el nombre (si lo tiene) del que envio el ICMP, mas el RTT de cada respuesta.
    print("{} ({})\t\t{}\n".format(
        IP_RECEIVE[0], IP_RECEIVE_NAME, [str(n) + " ms" for n in timer]
    ))
```

## VARIABLES GLOBALES:

IP\_ADDRESS = (192.168.0.2, 33434)

IP\_RECEIV = (192.168.122.2,0)

IP\_RECEIVE\_NAME = "\_gateway"

ACT\_TTL = 1

## SOCKETS:

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
0.32	n Bytes		

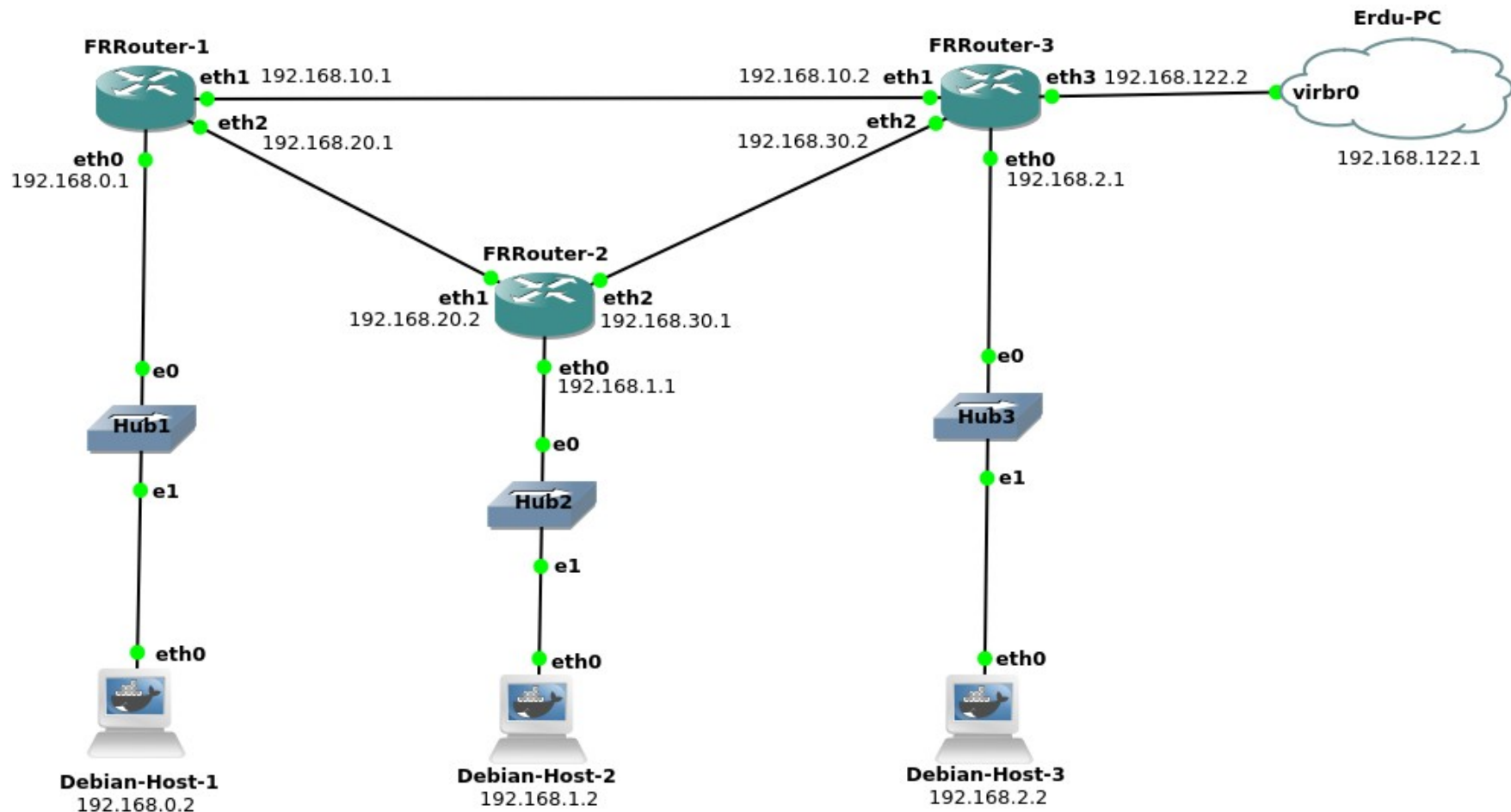
Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP



# Python3 tracert.py -a 192.168.0.2

```
erdu@PC-Erdu:~/Documentos/Python/tracert$ sudo python3 tracert.py -a 192.168.0.2
Trazando ruta hacia 192.168.0.2 (192.168.0.2) al puerto 33434 con 30 saltos maximos
> TTL = 1      192.168.122.2 ( gateway)      ['0.61 ms', '0.38 ms', '0.32 ms']
```





# Python3 tracer.py -a 192.168.0.2



## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV =  
(192.168.122.2, 0)

ACT\_TTL = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
0.32	n Bytes		



## SOCKETS:

Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP

```
#Imprimiendo informe avanzado
if (ADVANCED_MODE == True and len(packets) > 0):
    print("    Paquetes de respuesta (IP):")
    #Recorriendo la lista que contiene cada paquete IP recibido anteriormente.
    for packet in packets:
        ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
        #separar cada campo del paquete IP.

        #Imprimiendo campos mas relevantes del paquete IP actual.
        print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud trama = {} , "
              "DF = {} , MF = {} , TTL = {}".format(
                packets.index(packet) + 1,
                ip.getSourceAddress(),
                ip.getDestinationAddress(),
                ip.version,
                ip.identificador,
                ip.longitud_trama,
                ip.DontFragment,
                ip.MoreFragment,
                ip.TTL
            ))

        #Guardando en la lista, el paquete ICMP contenido dentro de este paquete IP.
        packets_icmp.append(ip.Data)

print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
```



# Python3 tracer.py -a 192.168.0.2



## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV =  
(192.168.122.2, 0)

ACT\_TTL = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
0.32	n Bytes		



## SOCKETS:

Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP

```
#Imprimiendo informe avanzado
if (ADVANCED_MODE == True and len(packets) > 0):
    print("    Paquetes de respuesta (IP):")
    #Recorriendo la lista que contiene cada paquete IP recibido anteriormente.
    for packet in packets:
        ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
        #separar cada campo del paquete IP.

        #Imprimiendo campos mas relevantes del paquete IP actual.
        print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud trama = {} , "
              "DF = {} , MF = {} , TTL = {}".format(
                packets.index(packet) + 1,
                ip.getSourceAddress(),
                ip.getDestinationAddress(),
                ip.version,
                ip.identificador,
                ip.longitud_trama,
                ip.DontFragment,
                ip.MoreFragment,
                ip.TTL
            ))

        #Guardando en la lista, el paquete ICMP contenido dentro de este paquete IP.
        packets_icmp.append(ip.Data)

print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
```



# Python3 tracer.py -a 192.168.0.2



## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV =  
(192.168.122.2, 0)

ACT\_TTL = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
0.32	n Bytes		



## SOCKETS:

Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP

```
#Imprimiendo informe avanzado
if (ADVANCED_MODE == True and len(packets) > 0):
    print("    Paquetes de respuesta (IP):")
    #Recorriendo la lista que contiene cada paquete IP recibido anteriormente.
    for packet in packets:
        ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
                               #separar cada campo del paquete IP.

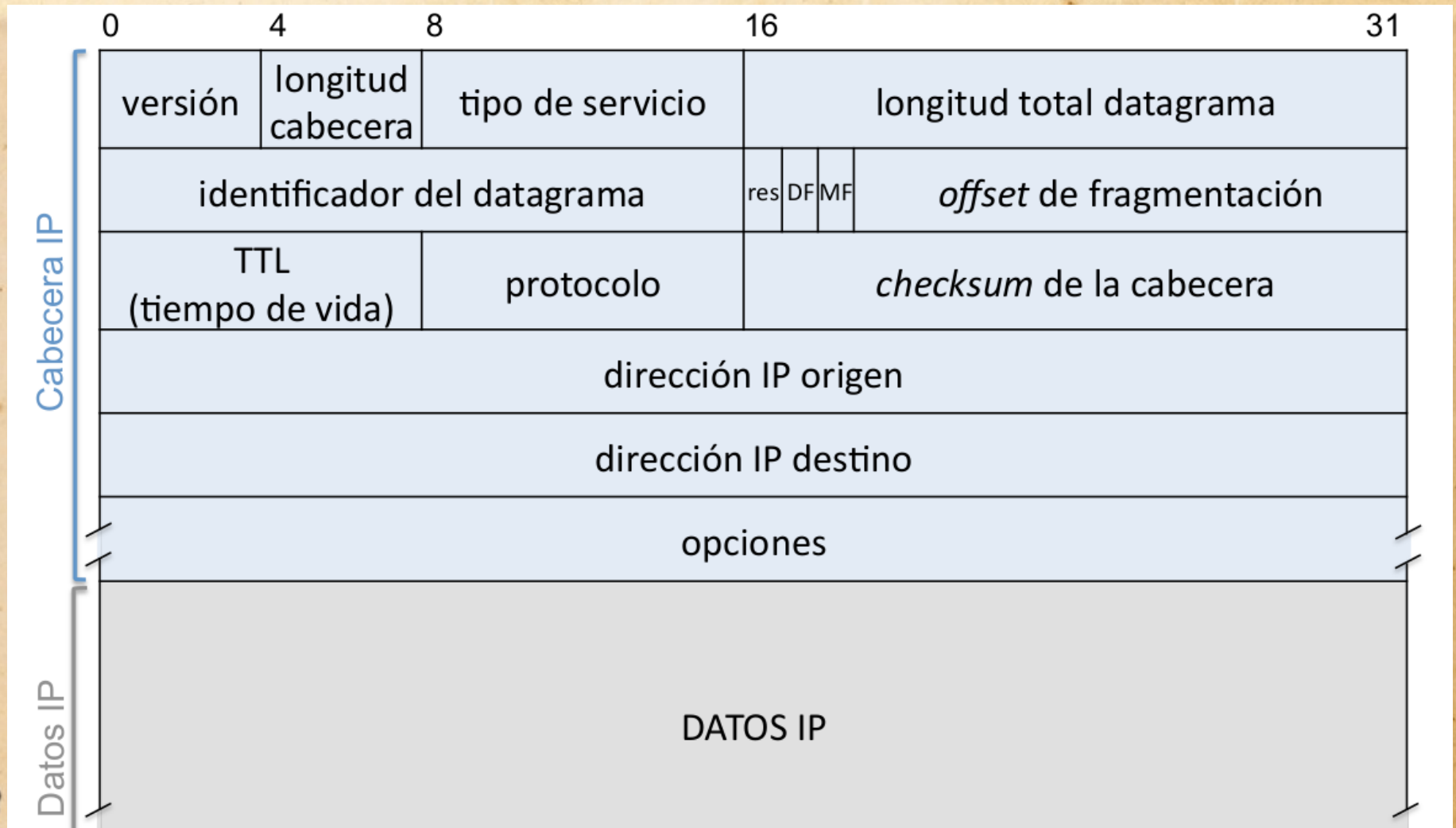
        #Imprimiendo campos mas relevantes del paquete IP actual.
        print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud trama = {} , "
              "DF = {} , MF = {} , TTL = {}".format(
                packets.index(packet) + 1,
                ip.getSourceAddress(),
                ip.getDestinationAddress(),
                ip.version,
                ip.identificador,
                ip.longitud_trama,
                ip.DontFragment,
                ip.MoreFragment,
                ip.TTL
            ))

        #Guardando en la lista, el paquete ICMP contenido dentro de este paquete IP.
        packets_icmp.append(ip.Data)

print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
```



# Campos del protocolo IP





# Nuevo Obj. “IPPacket”

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---

```
import sys, socket
```

```
class IP(object):
```

```
    def __init__(self, IPPacket = bytes()):
        self.version = IPPacket[0] >> 4 # 4 bits
        self.longitud_head = (IPPacket[0] & 15) # 4 bits
        self.servicio = IPPacket[1] # 1 Byte -> 8 bits
        self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
        self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
        self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
        self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
        self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
        self.TTL = IPPacket[8] # 1 Byte -> 8 bits
        self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
        self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
        self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                               (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
        self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                                   (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

```
#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP
```

```
#DATOS UTILES
```

```
self.Data = IPPacket[20:]
```

```
def getSourceAddress(self):
```

```
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
```

```
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100



# Nuevo Obj. “IPPacket”

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---

>> 4

```
import sys, socket
```

```
class IP(object):
```

```
    def __init__(self, IPPacket = bytes()):
        self.version = IPPacket[0] >> 4 # 4 bits
        self.longitud_head = (IPPacket[0] & 15) # 4 bits
        self.servicio = IPPacket[1] # 1 Byte -> 8 bits
        self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
        self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
        self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
        self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
        self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
        self.TTL = IPPacket[8] # 1 Byte -> 8 bits
        self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
        self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
        self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                               (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
        self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                                   (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

```
#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP
```

```
#DATOS UTILES
```

```
self.Data = IPPacket[20:]
```

```
def getSourceAddress(self):
```

```
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
```

```
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100



# Nuevo Obj. “IPPacket”

IPPacket[0] = 01000101

				0	1	0	0
--	--	--	--	---	---	---	---

```
import sys, socket
```

```
class IP(object):
```

```
    def __init__(self, IPPacket = bytes()):
        self.version = IPPacket[0] >> 4 # 4 bits
        self.longitud_head = (IPPacket[0] & 15) # 4 bits
        self.servicio = IPPacket[1] # 1 Byte -> 8 bits
        self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
        self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
        self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
        self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
        self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
        self.TTL = IPPacket[8] # 1 Byte -> 8 bits
        self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
        self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
        self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                               (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
        self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                                   (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

```
#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP
```

```
#DATOS UTILES
```

```
self.Data = IPPacket[20:]
```

```
def getSourceAddress(self):
```

```
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
```

```
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100



# Nuevo Obj. “IPPacket”

IPPacket[0] = 01000101

0	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---

```
import sys, socket
```

```
class IP(object):
```

```
    def __init__(self, IPPacket = bytes()):
        self.version = IPPacket[0] >> 4 # 4 bits
        self.longitud_head = (IPPacket[0] & 15) # 4 bits
        self.servicio = IPPacket[1] # 1 Byte -> 8 bits
        self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
        self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
        self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
        self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
        self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
        self.TTL = IPPacket[8] # 1 Byte -> 8 bits
        self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
        self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
        self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                               (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
        self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                                   (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

```
#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP
```

```
#DATOS UTILES
```

```
self.Data = IPPacket[20:]
```

```
def getSourceAddress(self):
```

```
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
```

```
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

IPPacket[0] = 01000101



0	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---

```
import sys, socket
```

```
class IP(object):
```

```
    def __init__(self, IPPacket = bytes()):
        self.version = IPPacket[0] >> 4 # 4 bits
        self.longitud_head = (IPPacket[0] & 15) # 4 bits
        self.servicio = IPPacket[1] # 1 Byte -> 8 bits
        self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
        self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
        self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
        self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
        self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
        self.TTL = IPPacket[8] # 1 Byte -> 8 bits
        self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
        self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
        self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                               (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
        self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                                   (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

```
#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP
```

```
#DATOS UTILES
```

```
self.Data = IPPacket[20:]
```

```
def getSourceAddress(self):
```

```
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
```

```
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

```
def __init__(self, IPPacket = bytes()):
    self.version = IPPacket[0] >> 4 # 4 bits
    self.longitud_head = (IPPacket[0] & 15) # 4 bits
    self.servicio = IPPacket[1] # 1 Byte -> 8 bits
    self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
    self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
    self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
    self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
    self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
    self.TTL = IPPacket[8] # 1 Byte -> 8 bits
    self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
    self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
    self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
        (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
    self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```

Packets[0]		
01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100



# Nuevo Obj. “IPPacket”

Version = 00000100 (4)

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---

Packets[0]		
01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
def __init__(self, IPPacket = bytes()):
    self.version = IPPacket[0] >> 4 # 4 bits
    self.longitud_head = (IPPacket[0] & 15) # 4 bits
    self.servicio = IPPacket[1] # 1 Byte -> 8 bits
    self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
    self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
    self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
    self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
    self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
    self.TTL = IPPacket[8] # 1 Byte -> 8 bits
    self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
    self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
    self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
        (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
    self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits

    #FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

    #DATOS UTILES
    self.Data = IPPacket[20:]

def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))

def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---

 = 69

AND

0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

 = 15

--	--	--	--	--	--	--	--

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---

 = 69

AND

0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

 = 15

0							
---	--	--	--	--	--	--	--

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1	= 69
AND								
0	0	0	0	1	1	1	1	= 15
0	0							

Packets[0]		
01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1	= 69
AND								
0	0	0	0	1	1	1	1	= 15
0	0	0						

Packets[0]		
01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---

 = 69

AND

0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

 = 15

0	0	0	0				
---	---	---	---	--	--	--	--

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                    (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1	= 69
AND								
0	0	0	0	1	1	1	1	= 15
0	0	0	0	0				

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---

 = 69

AND

0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

 = 15

0	0	0	0	0	1		
---	---	---	---	---	---	--	--

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                    (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---

 = 69

AND

0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

 = 15

0	0	0	0	0	1	0	
---	---	---	---	---	---	---	--

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1
AND							
0	0	0	0	1	1	1	1
0	0	0	0	0	1	0	1

= 69

= 15

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                    (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

IPPacket[0] = 01000101

0	1	0	0	0	1	0	1	= 69
AND								
0	0	0	0	1	1	1	1	= 15
0	0	0	0	0	1	0	1	= 5

Packets[0]		
01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)



0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

 = 5

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

110 = prioridad (control de red)

0 = retardo normal

0 = rendimiento normal

0 = fiabilidad normal

00 = no usados...

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)



## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)



Packets[0]		
01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)



Packets[0]		
01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)



Packets[0]		
01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. “IPPacket”

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)


$$+ \quad 00\underline{111111}$$

# Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
| | | | | (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
| | | | | (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

## #FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

## #DATOS UTILES

```
self.Data = IPPacket[20:]
```

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
16									9	8	1				

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                    (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. datagrama = 00000000 00111111 (63)

0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
16									9		8	1				

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. datagrama = 00000000 00111111 (63)

Cabecera + Datos = 63 Bytes

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.version = IPPacket[0] >> 4 # 4 bits
self.longitud_head = (IPPacket[0] & 15) # 4 bits
self.servicio = IPPacket[1] # 1 Byte -> 8 bits
self.longitud_trama = (IPPacket[2] << 8) + IPPacket[3] # 2 Bytes -> 16 bits
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 0

AND

0	1	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 64

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 0

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                    (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)


Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

= 0

 >> 6

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

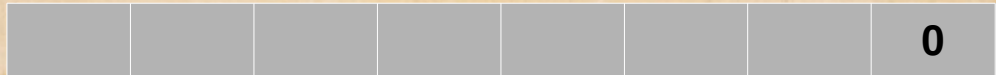
Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

 = 0

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Version = 00000100 (4)

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 0

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
```

```
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
```

```
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 0

AND

0	0	1	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 32

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 0

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                    (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

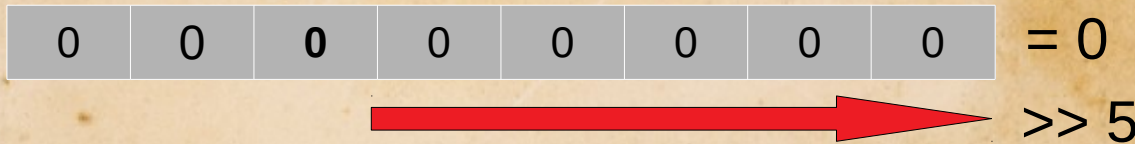
Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)



## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

							0
--	--	--	--	--	--	--	---

 = 0

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Longitud de cabecera = 00000101 (5)

Tipo de servicio = 11000000 (192)

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 0

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
```

```
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
```

```
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

= 0

AND

0	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---

= 31

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

= 0

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                    (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

0 0 0 0 0 0 0 0 = 0

<< 8



## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

0	0	0	0	0	0	0	0								
---	---	---	---	---	---	---	---	--	--	--	--	--	--	--	--

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
```

```
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
```

```
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

0	0	0	0	0	0	0	0								
---	---	---	---	---	---	---	---	--	--	--	--	--	--	--	--

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
```

```
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
```

```
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

Offset = 0

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Nuevo Obj. "IPPacket"

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

Offset = 00000000 00000000 (0)

TTL = 01000000 (64)

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

Offset = 00000000 00000000 (0)

TTL = 01000000 (64)

Protocolo = 00000001 (ICMP)

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Long. Datagrama = 00000000 00111111 (63)

Identificador = 01111100 10011001 (31897)

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

Offset = 00000000 00000000 (0)

TTL = 01000000 (64)

Protocolo = 00000001 (ICMP)

Checksum = 10001000 00010000

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

Don't Fragment = 00000000 (0)

More Fragment = 00000000 (0)

Offset = 00000000 00000000 (0)

TTL = 01000000 (64)

Protocolo = 00000001 (ICMP)

Checksum = 10001000 00010000

192

168

122

2

11000000.10101000.01111010.00000010

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                        (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

TTL = 01000000 (64)

Protocolo = 00000001 (ICMP)

Checksum = 10001000 00010000

IP Origen 192.168.122.2

192

168

122

2

11000000.10101000.01111010.00000010

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

TTL = 01000000 (64)

Protocolo = 00000001 (ICMP)

Checksum = 10001000 00010000

IP Origen = 192.168.122.2

192

168

122

2

11000000.10101000.01111010.00000001

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

TTL = 01000000 (64)

Protocolo = 00000001 (ICMP)

Checksum = 10001000 00010000

IP Origen = 192.168.122.2

IP Destino = 192.168.122.1

192

168

122

1

11000000.10101000.01111010.00000001

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

def getSourceAddress(self):

return socket.inet\_ntoa(self.SourceAddress.to\_bytes(4, 'big'))

def getDestinationAddress(self):

return socket.inet\_ntoa(self.DestinationAddress.to\_bytes(4, 'big'))



# Nuevo Obj. "IPPacket"

TTL = 01000000 (64)

Protocolo = 00000001 (ICMP)

Checksum = 10001000 00010000

IP Origen = 192.168.122.2

IP Destino = 192.168.122.1

## Packets[0]

01000101	11000000	00000000
00111111	01111100	10011001
00000000	00000000	01000000
00000001	10001000	00010000
11000000	10101000	01111010
00000010	11000000	10101000
01111010	00000001	00001011
00000000	11110000	01110100
00000000	00000000	00000000
00000000	01000101	00000000
00000000	00100011	01101010
10011001	01000000	00000000
00000001	00010001	00010011
11011101	11000000	10101000
01111010	00000001	11000000
10101000	00000000	00000010
10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100

```
self.identificador = (IPPacket[4] << 8) + IPPacket[5] # 2 Bytes -> 16 bits
self.DontFragment = (IPPacket[6] & 64) >> 6 # 1 bit
self.MoreFragment = (IPPacket[6] & 32) >> 5 # 1 bit
self.OffsetFragment = ((IPPacket[6] & 31) << 8) + IPPacket[7] # 13 bits
self.TTL = IPPacket[8] # 1 Byte -> 8 bits
self.Protocol = IPPacket[9] # 1 Byte -> 8 bits
self.Checksum = (IPPacket[10] << 8) + IPPacket[11] # 2 Bytes -> 16 bits
self.SourceAddress = (IPPacket[12] << 24) + (IPPacket[13] << 16) +
                    (IPPacket[14] << 8) + IPPacket[15] # 4 Bytes -> 32 bits
self.DestinationAddress = (IPPacket[16] << 24) + (IPPacket[17] << 16) +
                          (IPPacket[18] << 8) + IPPacket[19] # 4 Bytes -> 32 bits
```

#FALTA GESTIONAR CAMPO DE OPCIONES DE LA CABECERA DEL DATAGRAMA IP

#DATOS UTILES

self.Data = IPPacket[20:]

```
def getSourceAddress(self):
    return socket.inet_ntoa(self.SourceAddress.to_bytes(4, 'big'))
```

```
def getDestinationAddress(self):
    return socket.inet_ntoa(self.DestinationAddress.to_bytes(4, 'big'))
```



# Python3 tracer.py -a 192.168.0.2



```
#Imprimiendo informe avanzado
if (ADVANCED_MODE == True and len(packets) > 0):
    print("    Paquetes de respuesta (IP):")
    #Recorriendo la lista que contiene cada paquete IP recibido anteriormente.
    for packet in packets:
        ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
        #separar cada campo del paquete IP.

        #Imprimiendo campos mas relevantes del paquete IP actual.
        print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud trama = {} , "
              "DF = {} , MF = {} , TTL = {}".format(
                packets.index(packet) + 1,
                ip.getSourceAddress(),
                ip.getDestinationAddress(),
                ip.version,
                ip.identificador,
                ip.longitud_trama,
                ip.DontFragment,
                ip.MoreFragment,
                ip.TTL
            ))

        #Guardando en la lista, el paquete ICMP contenido dentro de este paquete IP.
        packets_icmp.append(ip.Data)

print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
```

## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV =  
(192.168.122.2,0)

ACT\_TTL = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
0.32	n Bytes		

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2



## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV =  
(192.168.122.2, 0)

ACT\_TTL = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes		
0.38	n Bytes		
0.32	n Bytes		

```
#Imprimiendo informe avanzado
if (ADVANCED_MODE == True and len(packets) > 0):
    print("    Paquetes de respuesta (IP):")
    #Recorriendo la lista que contiene cada paquete IP recibido anteriormente.
    for packet in packets:
        ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
        #separar cada campo del paquete IP.

        #Imprimiendo campos mas relevantes del paquete IP actual.
        print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud trama = {} , "
              "DF = {} , MF = {} , TTL = {}".format(
                packets.index(packet) + 1,
                ip.getSourceAddress(),
                ip.getDestinationAddress(),
                ip.version,
                ip.identificador,
                ip.longitud_trama,
                ip.DontFragment,
                ip.MoreFragment,
                ip.TTL
            ))

        #Guardando en la lista, el paquete ICMP contenido dentro de este paquete IP.
        packets_icmp.append(ip.Data)

print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
```

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2



## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV =  
(192.168.122.2, 0)

ACT\_TTL = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	N Bytes	
0.38	n Bytes		
0.32	n Bytes		

```
#Imprimiendo informe avanzado
if (ADVANCED_MODE == True and len(packets) > 0):
    print("    Paquetes de respuesta (IP):")
    #Recorriendo la lista que contiene cada paquete IP recibido anteriormente.
    for packet in packets:
        ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
        #separar cada campo del paquete IP.

        #Imprimiendo campos mas relevantes del paquete IP actual.
        print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud trama = {} , "
              "DF = {} , MF = {} , TTL = {}".format(
                packets.index(packet) + 1,
                ip.getSourceAddress(),
                ip.getDestinationAddress(),
                ip.version,
                ip.identificador,
                ip.longitud_trama,
                ip.DontFragment,
                ip.MoreFragment,
                ip.TTL
            ))

        #Guardando en la lista, el paquete ICMP contenido dentro de este paquete IP.
        packets_icmp.append(ip.Data)

print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
```

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2



## VARIABLES GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIV =  
(192.168.122.2, 0)

ACT\_TTL = 1

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	n Bytes	
0.38	n Bytes	n Bytes	
0.32	n Bytes	n Bytes	

## SOCKETS:

Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP

```
#Imprimiendo informe avanzado
if (ADVANCED_MODE == True and len(packets) > 0):
    print("    Paquetes de respuesta (IP):")
    #Recorriendo la lista que contiene cada paquete IP recibido anteriormente.
    for packet in packets:
        ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
        #separar cada campo del paquete IP.

        #Imprimiendo campos mas relevantes del paquete IP actual.
        print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud trama = {} , "
              "DF = {} , MF = {} , TTL = {}".format(
                packets.index(packet) + 1,
                ip.getSourceAddress(),
                ip.getDestinationAddress(),
                ip.version,
                ip.identificador,
                ip.longitud_trama,
                ip.DontFragment,
                ip.MoreFragment,
                ip.TTL
            ))

        #Guardando en la lista, el paquete ICMP contenido dentro de este paquete IP.
        packets_icmp.append(ip.Data)

    print("\n    Paquetes de respuesta (IP => ICMP):")
    packets.clear()
```



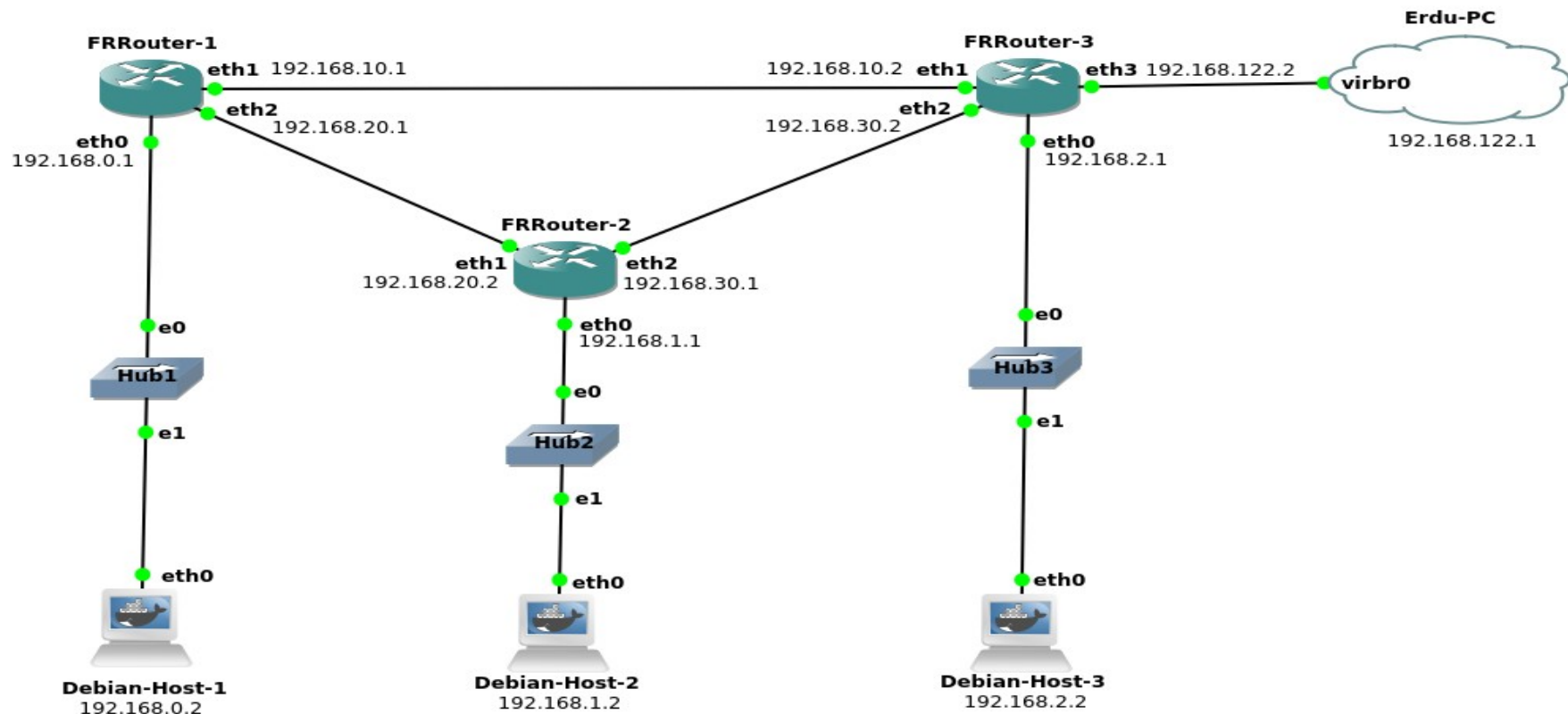
# Python3 tracert.py -a 192.168.0.2

```
erdu@PC-Erdu:~/Documentos/Python/tracert$ sudo python3 tracert.py -a 192.168.0.2
Trazando ruta hacia 192.168.0.2 (192.168.0.2) al puerto 33434 con 30 saltos maximos
```

```
> TTL = 1      192.168.122.2 ( gateway)      ['0.61 ms', '0.38 ms', '0.32 ms']
```

Paquetes de respuesta (IP):

1	( 192.168.122.2 => 192.168.122.1 )	Version = 4 , ID = 60805 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64
2	( 192.168.122.2 => 192.168.122.1 )	Version = 4 , ID = 60806 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64
3	( 192.168.122.2 => 192.168.122.1 )	Version = 4 , ID = 60807 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64





# Python3 tracer.py -a 192.168.0.2



```
print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
#Recorriendo la lista de paquetes ICMP que estaban contenidos en los paquetes IP recibidos.
for packet in packets_icmp:
    icmp = ICMPPacket(packet) #Obteniendo un objeto de la clase ICMP, que se encarga de
    #separar cada campo del paquete ICMP.

    #Imprimiendo campos mas relevantes del paquete ICMP.
    print("    {} | Tipo = {} ,Codigo = {}".format(
        packets_icmp.index(packet) + 1,
        icmp.Type,
        icmp.Code
    ))

    #Guardando en una lista los fragmentos de los paquetes IP originales.
    packets.append(icmp.Data)

print("\n    Paquetes de respuesta (IP => ICMP => IP):")
#Recorriendo la lista de los paquetes IP originales, que contenian los datagramas UDP
#que se enviaron anteriormente.
```

**VARIABLES GLOBALES:**  
IP\_ADDRESS =  
(192.168.0.2, 33434)

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	n Bytes	
0.38	n Bytes	n Bytes	
0.32	n Bytes	n Bytes	

## SOCKETS:

Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP



# Python3 tracer.py -a 192.168.0.2



```
print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
#Recorriendo la lista de paquetes ICMP que estaban contenidos en los paquetes IP recibidos.
for packet in packets_icmp:
    icmp = ICMPPacket(packet) #Obteniendo un objeto de la clase ICMP, que se encarga de
                               #separar cada campo del paquete ICMP.

    #Imprimiendo campos mas relevantes del paquete ICMP.
    print("    {} | Tipo = {} ,Codigo = {}".format(
        packets_icmp.index(packet) + 1,
        icmp.Type,
        icmp.Code
    ))

    #Guardando en una lista los fragmentos de los paquetes IP originales.
    packets.append(icmp.Data)

print("\n    Paquetes de respuesta (IP => ICMP => IP):")
#Recorriendo la lista de los paquetes IP originales, que contenian los datagramas UDP
#que se enviaron anteriormente.
```

**VARIABLES GLOBALES:**  
IP\_ADDRESS =  
(192.168.0.2, 33434)

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61		n Bytes	
0.38		n Bytes	
0.32		n Bytes	

## SOCKETS:

Socket RAW	Protocolo	IP	Puerto	Time Out
	ICMP	-	33434	5 seg

Socket UDP	Puerto	TTL	Protocolo
	-	1	UDP



# Nuevo Obj. “ICMPPacket”

```
import sys, socket

class ICMP(object):

    def __init__(self, Packet = bytes()):
        self.Type = Packet[0] # 1 Byte -> 8 bits
        self.Code = Packet[1] # 1 Byte -> 8 bits
        self.Checksum = (Packet[2] << 8) + Packet[3] # 2 Bytes -> 16 bits
        self.Data = Packet[8:] # N Bytes
```

Tipo = 00001011 (11) → TTL Excedido

## Packets\_icmp[0]

00001011	00000000	11110000
01110100	00000000	00000000
00000000	00000000	01000101
00000000	00000000	00100011
01101010	10011001	01000000
00000000	00000001	00010001
00010011	11011101	11000000
10101000	01111010	00000001
11000000	10101000	00000000
00000010	10001011	00110100
10000010	10011010	00000000
00001111	01100111	01100100
01010100	01110010	01100001
01100011	01100101	01110010
01110100		



# Nuevo Obj. “ICMPPacket”

```
import sys, socket

class ICMP(object):

    def __init__(self, Packet = bytes()):
        self.Type = Packet[0] # 1 Byte -> 8 bits
        self.Code = Packet[1] # 1 Byte -> 8 bits
        self.Checksum = (Packet[2] << 8) + Packet[3] # 2 Bytes -> 16 bits
        self.Data = Packet[8:] # N Bytes
```

Tipo = 00001011 (11)  
Codigo = 00000000 (0)

## Packets\_icmp[0]

00001011	00000000	11110000
01110100	00000000	00000000
00000000	00000000	01000101
00000000	00000000	00100011
01101010	10011001	01000000
00000000	00000001	00010001
00010011	11011101	11000000
10101000	01111010	00000001
11000000	10101000	00000000
00000010	10001011	00110100
10000010	10011010	00000000
00001111	01100111	01100100
01010100	01110010	01100001
01100011	01100101	01110010
01110100		



# Nuevo Obj. “ICMPPacket”

```
import sys, socket

class ICMP(object):

    def __init__(self, Packet = bytes()):
        self.Type = Packet[0] # 1 Byte -> 8 bits
        self.Code = Packet[1] # 1 Byte -> 8 bits
        self.Checksum = (Packet[2] << 8) + Packet[3] # 2 Bytes -> 16 bits
        self.Data = Packet[8:] # N Bytes
```

Tipo = 00001011 (11)

Codigo = 00000000 (0)

Tipo 11 y Codigo 0 = TTL Excedido.

## Packets\_icmp[0]

00001011	00000000	11110000
01110100	00000000	00000000
00000000	00000000	01000101
00000000	00000000	00100011
01101010	10011001	01000000
00000000	00000001	00010001
00010011	11011101	11000000
10101000	01111010	00000001
11000000	10101000	00000000
00000010	10001011	00110100
10000010	10011010	00000000
00001111	01100111	01100100
01010100	01110010	01100001
01100011	01100101	01110010
01110100		



# Nuevo Obj. “ICMPPacket”

```
import sys, socket

class ICMP(object):

    def __init__(self, Packet = bytes()):
        self.Type = Packet[0] # 1 Byte -> 8 bits
        self.Code = Packet[1] # 1 Byte -> 8 bits
        self.Checksum = (Packet[2] << 8) + Packet[3] # 2 Bytes -> 16 bits
        self.Data = Packet[8:] # N Bytes
```

Tipo = 00001011 (11)

Codigo = 00000000 (0).

Checksum = 11110000 01110100

## Packets\_icmp[0]

00001011	00000000	11110000
01110100	00000000	00000000
00000000	00000000	01000101
00000000	00000000	00100011
01101010	10011001	01000000
00000000	00000001	00010001
00010011	11011101	11000000
10101000	01111010	00000001
11000000	10101000	00000000
00000010	10001011	00110100
10000010	10011010	00000000
00001111	01100111	01100100
01010100	01110010	01100001
01100011	01100101	01110010
01110100		



# Nuevo Obj. “ICMPPacket”

```
import sys, socket

class ICMP(object):

    def __init__(self, Packet = bytes()):
        self.Type = Packet[0] # 1 Byte -> 8 bits
        self.Code = Packet[1] # 1 Byte -> 8 bits
        self.Checksum = (Packet[2] << 8) + Packet[3] # 2 Bytes -> 16 bits
        self.Data = Packet[8:] # N Bytes
```

Tipo = 00001011 (11)

Codigo = 00000000 (0)

Checksum = 11110000 01110100

4 Bytes (32 bits) no utilizados.

## Packets\_icmp[0]

00001011	00000000	11110000
01110100	00000000	00000000
00000000	00000000	01000101
00000000	00000000	00100011
01101010	10011001	01000000
00000000	00000001	00010001
00010011	11011101	11000000
10101000	01111010	00000001
11000000	10101000	00000000
00000010	10001011	00110100
10000010	10011010	00000000
00001111	01100111	01100100
01010100	01110010	01100001
01100011	01100101	01110010
01110100		



# Python3 tracer.py -a 192.168.0.2



```
print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
#Recorriendo la lista de paquetes ICMP que estaban contenidos en los paquetes IP recibidos.
for packet in packets_icmp:
    icmp = ICMPPacket(packet) #Obteniendo un objeto de la clase ICMP, que se encarga de
    #separar cada campo del paquete ICMP.

    #Imprimiendo campos mas relevantes del paquete ICMP.
    print("    {} | Tipo = {} ,Codigo = {}".format(
        packets_icmp.index(packet) + 1,
        icmp.Type,
        icmp.Code
    ))

    #Guardando en una lista los fragmentos de los paquetes IP originales.
    packets.append(icmp.Data)

print("\n    Paquetes de respuesta (IP => ICMP => IP):")
#Recorriendo la lista de los paquetes IP originales, que contenian los datagramas UDP
#que se enviaron anteriormente.
```

**VARIABLES GLOBALES:**  
IP\_ADDRESS =  
(192.168.0.2, 33434)

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61		n Bytes	
0.38		n Bytes	
0.32		n Bytes	

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracert.py -a 192.168.0.2

```
erdu@PC-Erdu:~/Documentos/Python/tracert$ sudo python3 tracert.py -a 192.168.0.2
Trazando ruta hacia 192.168.0.2 (192.168.0.2) al puerto 33434 con 30 saltos maximos
```

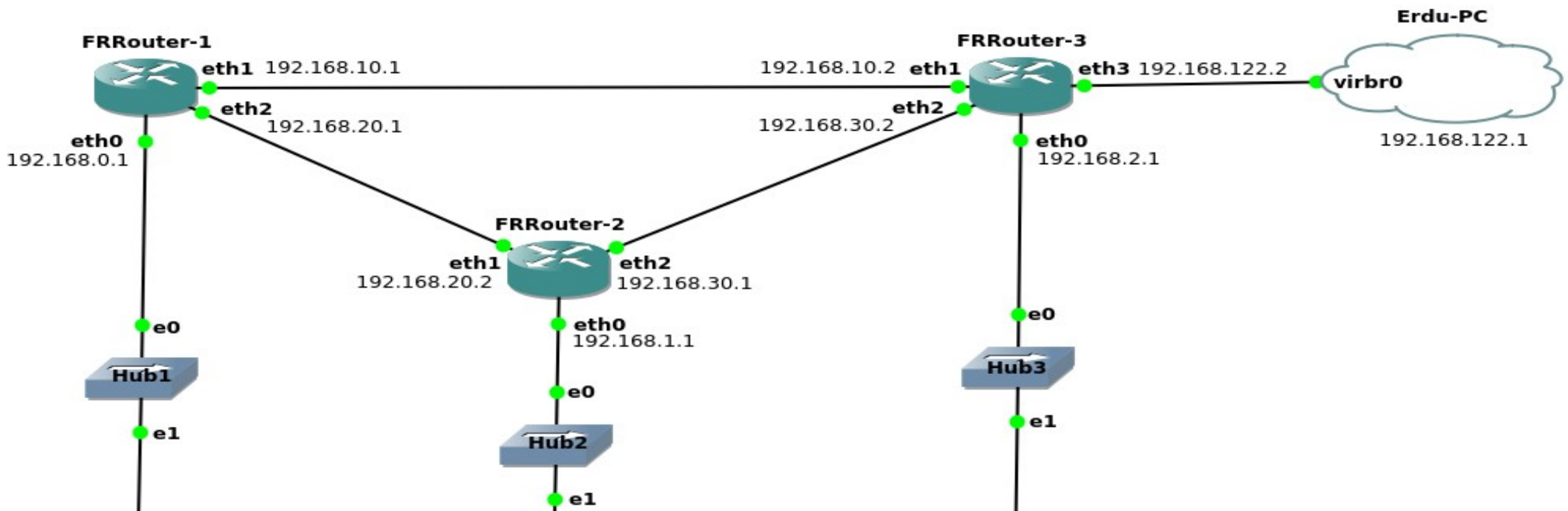
```
> TTL = 1      192.168.122.2 ( gateway)      ['0.61 ms', '0.38 ms', '0.32 ms']
```

Paquetes de respuesta (IP):

1	( 192.168.122.2 => 192.168.122.1 )	Version = 4 , ID = 60805 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64
2	( 192.168.122.2 => 192.168.122.1 )	Version = 4 , ID = 60806 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64
3	( 192.168.122.2 => 192.168.122.1 )	Version = 4 , ID = 60807 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64

Paquetes de respuesta (IP => ICMP):

1	Tipo = 11 ,Codigo = 0
2	Tipo = 11 ,Codigo = 0
3	Tipo = 11 ,Codigo = 0





# Python3 tracer.py -a 192.168.0.2



```
print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
#Recorriendo la lista de paquetes ICMP que estaban contenidos en los paquetes IP recibidos.
for packet in packets_icmp:
    icmp = ICMPPacket(packet) #Obteniendo un objeto de la clase ICMP, que se encarga de
    #separar cada campo del paquete ICMP.

    #Imprimiendo campos mas relevantes del paquete ICMP.
    print("    {} | Tipo = {} ,Codigo = {}".format(
        packets_icmp.index(packet) + 1,
        icmp.Type,
        icmp.Code
    ))

    #Guardando en una lista los fragmentos de los paquetes IP originales.
    packets.append(icmp.Data)

print("\n    Paquetes de respuesta (IP => ICMP => IP):")
#Recorriendo la lista de los paquetes IP originales, que contenian los datagramas UDP
#que se enviaron anteriormente.
```

**VARIABLES GLOBALES:**  
IP\_ADDRESS =  
(192.168.0.2, 33434)

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61		n Bytes	
0.38		n Bytes	
0.32		n Bytes	

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2



```
print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
#Recorriendo la lista de paquetes ICMP que estaban contenidos en los paquetes IP recibidos.
for packet in packets_icmp:
    icmp = ICMPPacket(packet) #Obteniendo un objeto de la clase ICMP, que se encarga de
    #separar cada campo del paquete ICMP.

    #Imprimiendo campos mas relevantes del paquete ICMP.
    print("    {} | Tipo = {} ,Codigo = {}".format(
        packets_icmp.index(packet) + 1,
        icmp.Type,
        icmp.Code
    ))

    #Guardando en una lista los fragmentos de los paquetes IP originales.
    packets.append(icmp.Data)

print("\n    Paquetes de respuesta (IP => ICMP => IP):")
#Recorriendo la lista de los paquetes IP originales, que contenian los datagramas UDP
#que se enviaron anteriormente.
```

**VARIABLES GLOBALES:**  
IP\_ADDRESS =  
(192.168.0.2, 33434)

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	n Bytes	
0.38		n Bytes	
0.32		n Bytes	

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2



```
print("\n    Paquetes de respuesta (IP => ICMP):")
packets.clear()
#Recorriendo la lista de paquetes ICMP que estaban contenidos en los paquetes IP recibidos.
for packet in packets_icmp:
    icmp = ICMPPacket(packet) #Obteniendo un objeto de la clase ICMP, que se encarga de
    #separar cada campo del paquete ICMP.

    #Imprimiendo campos mas relevantes del paquete ICMP.
    print("    {} | Tipo = {} ,Codigo = {}".format(
        packets_icmp.index(packet) + 1,
        icmp.Type,
        icmp.Code
    ))

    #Guardando en una lista los fragmentos de los paquetes IP originales.
    packets.append(icmp.Data)

print("\n    Paquetes de respuesta (IP => ICMP => IP):")
#Recorriendo la lista de los paquetes IP originales, que contenian los datagramas UDP
#que se enviaron anteriormente.
```

**VARIABLES GLOBALES:**  
IP\_ADDRESS =  
(192.168.0.2, 33434)

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	n Bytes	
0.38	n Bytes	n Bytes	
0.32	n Bytes	n Bytes	

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
###
print("\n    Paquetes de respuesta (IP => ICMP => IP):")
#Recorriendo la lista de los paquetes IP originales, que contenian los datagramas UDP
#que se enviaron anteriormente.
for packet in packets:
    ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
    #separar cada campo del paquete IP.

    #Imprimiendo campos mas relevantes del paquete IP actual.
    print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud datagrama = {} , DF = {} , MF = {} ,
          packets.index(packet) + 1,
          ip.getSourceAddress(),
          ip.getDestinationAddress(),
          ip.version,
          ip.identificador,
          ip.longitud_trama,
          ip.DontFragment,
          ip.MoreFragment,
          ip.TTL
    ))

    packets_udp.append(ip.Data)

###
print("\n    Paquetes de respuesta (IP => ICMP => IP => UDP):")
#Recorriendo la lista de los datagramas UDP originales, que contenian los paquetes IP originales.
```

**VARIABLES  
GLOBALES:**  
IP\_ADDRESS =  
(192.168.0.2,  
33434)

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	n Bytes	
0.38	n Bytes	n Bytes	
0.32	n Bytes	n Bytes	

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2

```
###
print("\n    Paquetes de respuesta (IP => ICMP => IP):")
#Recorriendo la lista de los paquetes IP originales, que contenian los datagramas UDP
#que se enviaron anteriormente.
for packet in packets:
    ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
    #separar cada campo del paquete IP.

    #Imprimiendo campos mas relevantes del paquete IP actual.
    print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud datagrama = {} , DF = {} , MF = {} ,
    packets.index(packet) + 1,
    ip.getSourceAddress(),
    ip.getDestinationAddress(),
    ip.version,
    ip.identificador,
    ip.longitud_trama,
    ip.DontFragment,
    ip.MoreFragment,
    ip.TTL
    ))

    packets_udp.append(ip.Data)

###
print("\n    Paquetes de respuesta (IP => ICMP => IP => UDP):")
#Recorriendo la lista de los datagramas UDP originales, que contenian los paquetes IP originales.
```

**VARIABLES  
GLOBALES:**  
IP\_ADDRESS =  
(192.168.0.2,  
33434)

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	n Bytes	
0.38	n Bytes	n Bytes	
0.32	n Bytes	n Bytes	

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracert.py -a 192.168.0.2

```
erdu@PC-Erdu:~/Documentos/Python/tracert$ sudo python3 tracert.py -a 192.168.0.2
Trazando ruta hacia 192.168.0.2 (192.168.0.2) al puerto 33434 con 30 saltos maximos
```

```
> TTL = 1      192.168.122.2 ( gateway)                ['0.61 ms', '0.38 ms', '0.32 ms']
```

Paquetes de respuesta (IP):

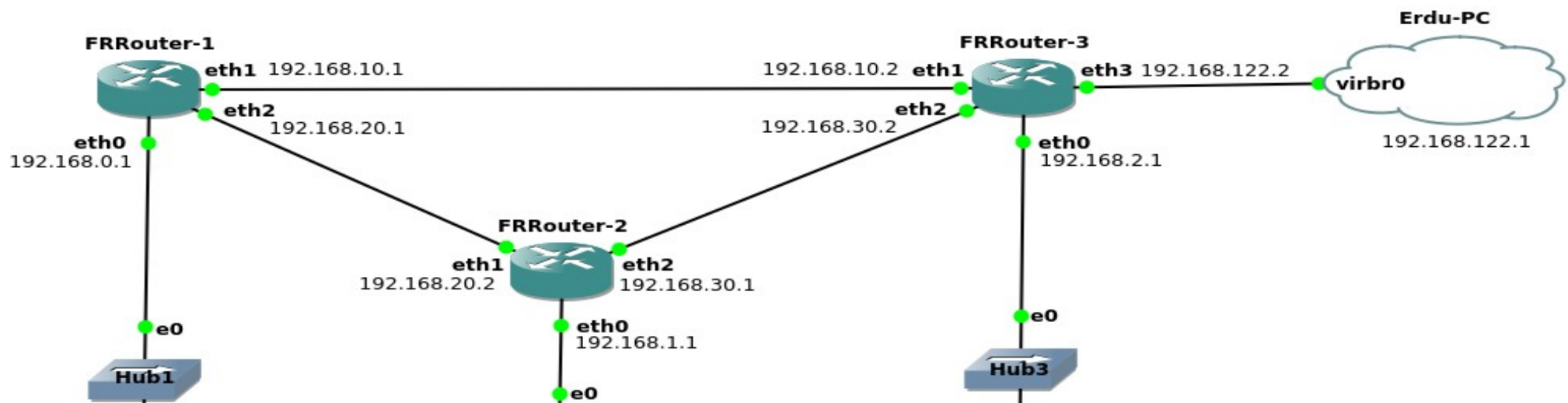
1   ( 192.168.122.2 => 192.168.122.1 )	Version = 4 , ID = 60805 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64
2   ( 192.168.122.2 => 192.168.122.1 )	Version = 4 , ID = 60806 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64
3   ( 192.168.122.2 => 192.168.122.1 )	Version = 4 , ID = 60807 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64

Paquetes de respuesta (IP => ICMP):

1   Tipo = 11 ,Codigo = 0
2   Tipo = 11 ,Codigo = 0
3   Tipo = 11 ,Codigo = 0

Paquetes de respuesta (IP => ICMP => IP):

1   ( 192.168.122.1 => 192.168.0.2 )	Version = 4 , ID = 1477 , Longitud trama = 35 , DF = 1 , MF = 0 , TTL = 1
2   ( 192.168.122.1 => 192.168.0.2 )	Version = 4 , ID = 1478 , Longitud trama = 35 , DF = 1 , MF = 0 , TTL = 1
3   ( 192.168.122.1 => 192.168.0.2 )	Version = 4 , ID = 1479 , Longitud trama = 35 , DF = 1 , MF = 0 , TTL = 1





# Python3 tracer.py -a 192.168.0.2



**VARIABLES  
GLOBALES:**  
IP\_ADDRESS =  
(192.168.0.2,  
33434)

```
###
print("\n    Paquetes de respuesta (IP => ICMP => IP):")
#Recorriendo la lista de los paquetes IP originales, que contenian los datagramas UDP
#que se enviaron anteriormente.
for packet in packets:
    ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
    #separar cada campo del paquete IP.

    #Imprimiendo campos mas relevantes del paquete IP actual.
    print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud datagrama = {} , DF = {} , MF = {} ,
          packets.index(packet) + 1,
          ip.getSourceAddress(),
          ip.getDestinationAddress(),
          ip.version,
          ip.identificador,
          ip.longitud_trama,
          ip.DontFragment,
          ip.MoreFragment,
          ip.TTL
    ))

    packets_udp.append(ip.Data)

###
print("\n    Paquetes de respuesta (IP => ICMP => IP => UDP):")
#Recorriendo la lista de los datagramas UDP originales, que contenian los paquetes IP originales.
```

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	n Bytes	
0.38	n Bytes	n Bytes	
0.32	n Bytes	n Bytes	

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2



**VARIABLES  
GLOBALES:**  
IP\_ADDRESS =  
(192.168.0.2,  
33434)

```
###
print("\n    Paquetes de respuesta (IP => ICMP => IP):")
#Recorriendo la lista de los paquetes IP originales, que contenian los datagramas UDP
#que se enviaron anteriormente.
for packet in packets:
    ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
    #separar cada campo del paquete IP.

    #Imprimiendo campos mas relevantes del paquete IP actual.
    print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud datagrama = {} , DF = {} , MF = {} ,
          packets.index(packet) + 1,
          ip.getSourceAddress(),
          ip.getDestinationAddress(),
          ip.version,
          ip.identificador,
          ip.longitud_trama,
          ip.DontFragment,
          ip.MoreFragment,
          ip.TTL
    ))

    packets_udp.append(ip.Data)

###
print("\n    Paquetes de respuesta (IP => ICMP => IP => UDP):")
#Recorriendo la lista de los datagramas UDP originales, que contenian los paquetes IP originales.
```

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	n Bytes	n Bytes
0.38	n Bytes	n Bytes	
0.32	n Bytes	n Bytes	

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2



**VARIABLES  
GLOBALES:**  
IP\_ADDRESS =  
(192.168.0.2,  
33434)

```
###
print("\n    Paquetes de respuesta (IP => ICMP => IP):")
#Recorriendo la lista de los paquetes IP originales, que contenian los datagramas UDP
#que se enviaron anteriormente.
for packet in packets:
    ip = IPPacket(packet) #Obteniendo un objeto de la clase IP, que se encarga de
    #separar cada campo del paquete IP.

    #Imprimiendo campos mas relevantes del paquete IP actual.
    print("    {} | ( {} => {} )\tVersion = {} , ID = {} , Longitud datagrama = {} , DF = {} , MF = {} ,
    packets.index(packet) + 1,
    ip.getSourceAddress(),
    ip.getDestinationAddress(),
    ip.version,
    ip.identificador,
    ip.longitud_trama,
    ip.DontFragment,
    ip.MoreFragment,
    ip.TTL
    ))

    packets_udp.append(ip.Data)

###
print("\n    Paquetes de respuesta (IP => ICMP => IP => UDP):")
#Recorriendo la lista de los datagramas UDP originales, que contenian los paquetes IP originales.
```

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	n Bytes	n Bytes
0.38	n Bytes	n Bytes	n Bytes
0.32	n Bytes	n Bytes	n Bytes

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

Socket	Puerto	TTL	Protocolo
UDP	-	1	UDP



# Python3 tracer.py -a 192.168.0.2



```
print("\n Paquetes de respuesta (IP => ICMP => IP => UDP):")
#Recorriendo la lista de los datagramas UDP originales, que contenian los paquetes IP originales.
for packet in packets_udp:
    udp_datagram = UDPPacket(packet) #Obteniendo un objeto de la clase UDP, que se encarga de
    #separar cada campo del datagrama UDP.

    #Imprimiendo campos mas relevantes del paquete IP actual.
    print(" {} | ( PO: {} => PD: {} )\tLongitud = {} , Datos = {}".format(
        packets_udp.index(packet) + 1,
        udp_datagram.SourcePort,
        udp_datagram.DestinationPort,
        udp_datagram.Lenght,
        str(udp_datagram.Data)[1:]
    ))

print("\n\n")

#Comprobando que todavia no hayamos llegado al destino
if len(packets_icmp) > 0:
    icmp = ICMPPacket(packets_icmp[0])
    if (icmp.Code == 3 and icmp.Type == 3):
        break

#Comprobando que no se hayan sobrepasado los saltos maximos.
if ACT_TTL >= MAX_HOPS and IP_ADDRESS[0] != IP_RECEIVE[0]:
    print("Saltos maximos alcanzados... Operacion cancelada...")
    break
```

## VARIABLES

### GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE =  
(192.168.122.2, 0)

MAX\_HOPS = 30  
ACT\_TTL = 1

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	n Bytes	n Bytes
0.38	n Bytes	n Bytes	n Bytes
0.32	n Bytes	n Bytes	n Bytes



# Nuevo Obj. “UDPPacket”

```
import sys, socket

class UDP(object):

    def __init__(self, Packet = bytes()):
        self.SourcePort = (Packet[0] << 8) + Packet[1] # 2 Bytes -> 16 bits
        self.DestinationPort = (Packet[2] << 8) + Packet[3] # 2 Bytes -> 16 bits
        self.Length = (Packet[4] << 8) + Packet[5] # 2 Bytes -> 16 bits
        self.Checksum = (Packet[6] << 8) + Packet[7] # 2 Bytes -> 16 bits
        self.Data = Packet[8:] # N Bytes
```

Puerto origen: 39047

## Packets\_udp[0]

10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100



# Nuevo Obj. “UDPPacket”

```
import sys, socket

class UDP(object):

    def __init__(self, Packet = bytes()):
        self.SourcePort = (Packet[0] << 8) + Packet[1] # 2 Bytes -> 16 bits
        self.DestinationPort = (Packet[2] << 8) + Packet[3] # 2 Bytes -> 16 bits
        self.Length = (Packet[4] << 8) + Packet[5] # 2 Bytes -> 16 bits
        self.Checksum = (Packet[6] << 8) + Packet[7] # 2 Bytes -> 16 bits
        self.Data = Packet[8:] # N Bytes
```

Puerto origen = 39047

Puerto destino = 33434

## Packets\_udp[0]

10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100



# Nuevo Obj. “UDPPacket”

```
import sys, socket

class UDP(object):

    def __init__(self, Packet = bytes()):
        self.SourcePort = (Packet[0] << 8) + Packet[1] # 2 Bytes -> 16 bits
        self.DestinationPort = (Packet[2] << 8) + Packet[3] # 2 Bytes -> 16 bits
        self.Length = (Packet[4] << 8) + Packet[5] # 2 Bytes -> 16 bits
        self.Checksum = (Packet[6] << 8) + Packet[7] # 2 Bytes -> 16 bits
        self.Data = Packet[8:] # N Bytes
```

Puerto origen = 39047

Puerto destino = 33434

Longitud = 15

## Packets\_udp[0]

10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100



# Nuevo Obj. “UDPPacket”

```
import sys, socket

class UDP(object):

    def __init__(self, Packet = bytes()):
        self.SourcePort = (Packet[0] << 8) + Packet[1] # 2 Bytes -> 16 bits
        self.DestinationPort = (Packet[2] << 8) + Packet[3] # 2 Bytes -> 16 bits
        self.Length = (Packet[4] << 8) + Packet[5] # 2 Bytes -> 16 bits
        self.Checksum = (Packet[6] << 8) + Packet[7] # 2 Bytes -> 16 bits
        self.Data = Packet[8:] # N Bytes
```

Puerto origen = 39047

Puerto destino = 33434

Longitud = 15

## Packets\_udp[0]

10001011	00110100	10000010
10011010	00000000	00001111
01100111	01100100	01010100
01110010	01100001	01100011
01100101	01110010	01110100



# Python3 tracer.py -a 192.168.0.2



```
print("\n Paquetes de respuesta (IP => ICMP => IP => UDP):")
#Recorriendo la lista de los datagramas UDP originales, que contenian los paquetes IP originales.
for packet in packets_udp:
    udp_datagram = UDPPacket(packet) #Obteniendo un objeto de la clase UDP, que se encarga de
    #separar cada campo del datagrama UDP.

    #Imprimiendo campos mas relevantes del paquete IP actual.
    print(" {} | ( PO: {} => PD: {} )\tLongitud = {} , Datos = {}".format(
        packets_udp.index(packet) + 1,
        udp_datagram.SourcePort,
        udp_datagram.DestinationPort,
        udp_datagram.Lenght,
        str(udp_datagram.Data)[1:]
    ))

print("\n\n")

#Comprobando que todavia no hayamos llegado al destino
if len(packets_icmp) > 0:
    icmp = ICMPPacket(packets_icmp[0])
    if (icmp.Code == 3 and icmp.Type == 3):
        break

#Comprobando que no se hayan sobrepasado los saltos maximos.
if ACT_TTL >= MAX_HOPS and IP_ADDRESS[0] != IP_RECEIVE[0]:
    print("Saltos maximos alcanzados... Operacion cancelada...")
    break
```

## VARIABLES

### GLOBALES:

IP\_ADDRESS =  
(192.168.0.2, 33434)

IP\_RECEIVE =  
(192.168.122.2, 0)

MAX\_HOPS = 30  
ACT\_TTL = 1

## SOCKETS:

Socket	Protocolo	IP	Puerto	Time Out
RAW	ICMP	-	33434	5 seg

## Listas:

Timer	packets	Packets_icmp	Packets_udp
0.61	n Bytes	n Bytes	n Bytes
0.38	n Bytes	n Bytes	n Bytes
0.32	n Bytes	n Bytes	n Bytes



```
erdu@PC-Erdu:~/Documentos/Python/tracert$ sudo python3 tracert-alt.py -a 192.168.0.2
Trazando ruta hacia 192.168.0.2 (192.168.0.2) al puerto 33434 con 30 saltos maximos
```

```
TI = 1571212036.4519813 | TF = 1571212036.4524698
TI = 1571212036.4526205 | TF = 1571212036.4529624
TI = 1571212036.4530149 | TF = 1571212036.4532862
> TTL = 1 192.168.122.2 ( gateway) [ '0.61 ms', '0.38 ms', '0.32 ms' ]
```

Paquetes de respuesta (IP):

```
1 | ( 192.168.122.2 => 192.168.122.1 ) Version = 4 , ID = 60805 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64
2 | ( 192.168.122.2 => 192.168.122.1 ) Version = 4 , ID = 60806 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64
3 | ( 192.168.122.2 => 192.168.122.1 ) Version = 4 , ID = 60807 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 64
```

Paquetes de respuesta (IP => ICMP):

```
1 | Tipo = 11 ,Codigo = 0
2 | Tipo = 11 ,Codigo = 0
3 | Tipo = 11 ,Codigo = 0
```

Paquetes de respuesta (IP => ICMP => IP):

```
1 | ( 192.168.122.1 => 192.168.0.2 ) Version = 4 , ID = 1477 , Longitud trama = 35 , DF = 1 , MF = 0 , TTL = 1
2 | ( 192.168.122.1 => 192.168.0.2 ) Version = 4 , ID = 1478 , Longitud trama = 35 , DF = 1 , MF = 0 , TTL = 1
3 | ( 192.168.122.1 => 192.168.0.2 ) Version = 4 , ID = 1479 , Longitud trama = 35 , DF = 1 , MF = 0 , TTL = 1
```

Paquetes de respuesta (IP => ICMP => IP => UDP):

```
1 | ( P0: 35812 => PD: 33434 ) Longitud = 15 , Datos = 'Tracert'
1 | ( P0: 35812 => PD: 33434 ) Longitud = 15 , Datos = 'Tracert'
1 | ( P0: 35812 => PD: 33434 ) Longitud = 15 , Datos = 'Tracert'
```

```
TI = 1571212036.456787 | TF = 1571212036.457341
TI = 1571212036.4574018 | TF = 1571212036.4576747
TI = 1571212036.4577205 | TF = 1571212036.4580474
> TTL = 2 192.168.10.1 (-) [ '0.6 ms', '0.31 ms', '0.38 ms' ]
```

Paquetes de respuesta (IP):

```
1 | ( 192.168.10.1 => 192.168.122.1 ) Version = 4 , ID = 55106 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 63
2 | ( 192.168.10.1 => 192.168.122.1 ) Version = 4 , ID = 55107 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 63
3 | ( 192.168.10.1 => 192.168.122.1 ) Version = 4 , ID = 55108 , Longitud trama = 63 , DF = 0 , MF = 0 , TTL = 63
```

Paquetes de respuesta (IP => ICMP):

```
1 | Tipo = 11 ,Codigo = 0
```





This work is licensed under  
a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
It makes use of the works of  
Kelly Loves Whales and Nick Merritt.