Memoria práctica 2 Redes de Comunicaciones

Miguel Jesús Paterson González
Mijaíl Sazhin Martín

Objetivos

Los objetivos de esta práctica han sido los siguientes: comprender el uso del protocolo ARP y su funcionalidad así como su implementación, manejo y su optimización a través de una caché.

Desglose del código

Las funciones desglosadas tendrán el mismo orden que el de aparición en los ficheros

ethernet.py

process_Ethernet_frame

```
def
process_Ethernet_frame(us:ctypes.c_void_p,header:pcap_pkthdr,data:bytes
) -> None:
    global macAddress
    destino = data[:6]
    origen = data[6:12]
    tipo = data[12:14]
    payload = data[14:]

    logging.debug(f"DESTINO: {destino} ORIGEN: {origen} TIPO: {tipo}
PAYLOAD: {payload}")

    if (destino == broadcastAddr or destino == macAddress): # Si la
dirección destino es la propia o la de broadcast
        if tipo in EthernetProtocols.keys():
            func = EthernetProtocols[tipo]
            func(us, header, payload, origen)
            return
    return
```

Esta función se encarga de procesar la trama Ethernet entrante. Se extraen la mac destino, la origen y el tipo de la cabecera y a continuación el payload. Después se comprueba si la mac destino es la mac de difusión (FF:FF:FF:FF) o si es la mac de la propia interfaz, después comprueba el Ethertype para ejecutar la función de callback correspondiente.

registerEthCallback

```
def registerEthCallback(callback_func:
Callable[[ctypes.c_void_p,pcap_pkthdr,bytes],None], ethertype:int) ->
None:

    global EthernetProtocols
    #EthernetProtocols es el diccionario que relaciona función de
callback y ethertype
    EthernetProtocols.update({ethertype.to_bytes(2, 'big'):
callback_func})
```

Esta función registra en el diccionario la función a ejecutar según los diferentes Ethertype a tratar

startEthernetLevel

```
def startEthernetLevel(interface:str) -> int:

    global macAddress,handle,levelInitialized,recvThread
    handle = None
    errbuf = bytearray()

    if globals().get('levelInitialized', False) == True: # Comprobamos
si el nivel Ethernet ya estaba inicializado
        return -1
    levelInitialized = False

    handle = pcap_open_live(interface, ETH_FRAME_MAX, PROMISC, TO_MS,
errbuf) # Abrimos la interfaz en modo promiscuo
    if not handle:
        return -1

    macAddress = getHwAddr(interface) # Obtenemos la dirección MAC
asociada a la interfaz

    #Una vez hemos abierto la interfaz para captura y hemos
inicializado las variables globales (macAddress, handle y
levelInitialized) arrancamos
```

```
    #el hilo de recepción

    recvThread = rxThread()
    recvThread.daemon = True
    recvThread.start()
    levelInitialized = True
    return 0
```

Esta función se encarga de inicializar el nivel Ethernet. Primero comprueba si se ha inicializado anteriormente devolviendo error si es el caso. Después abre la interfaz y obtiene la dirección MAC de la interfaz. Por último inicia el hilo de recepción para futuras requests.

stopEthernetLevel

```
def stopEthernetLevel()->int:
    global macAddress,handle,levelInitialized,recvThread
    try:
        pcap_close(handle)
        recvThread.stop()
    except:
        return -1

    levelInitialized = False
    return 0
```

Esta función se encarga de finalizar el nivel Ethernet cerrando la interfaz y finalizando el hilo

sendEthernetFrame

```
def sendEthernetFrame(data:bytes,length:int,etherType:int,dstMac:bytes)
-> int:

    global macAddress,handle

    if not data or not length or not etherType or not dstMac:
        return -1

    header = dstMac + macAddress + etherType.to_bytes(2, 'big') #
Construimos la cabecera de la trama Ethernet

    if (length < (ETH_FRAME_MIN - LEN_HEADER)): # Si la trama es menor
que el tamaño mínimo (60 bytes) rellenamos con 0s
        ceros = (ETH_FRAME_MIN - LEN_HEADER) - length
```

```
        data += b'\x00' * ceros
    elif (length > 1500): # Si la trama es mayor que el tamaño máximo
(1514 bytes) devolvemos error
        return -1

    data = header + data

    logging.debug(f"Trama a enviar: {data}")

    try:
        pcap_inject(handle, data, len(data))
    except:
        logging.error("Error al enviar la trama")
        return -1
    logging.debug("Trama enviada")
    return 0
```

Esta función se encargará de construir una trama Ethernet con los datos recibidos y lo enviará por la red. Primero construimos la cabecera Ethernet formada por la MAC destino, la MAC origen y el Ethertype. Después comprobamos si los datos recibidos cumplen con la longitud mínima (agregando 0s si no) y la máxima (devolviendo error). Por último se hará un pcap_inject para enviar la trama por la red.

ethmsg.py

process_ethMsg_frame

```
def
process_ethMsg_frame(us:ctypes.c_void_p,header:pcap_pkthdr,data:bytes,s
rcMac:bytes) -> None:

    if data is None:
        return

    message = f"[{header.ts.tv_sec}.{header.ts.tv_usec}] {srcMac} ->
{data[0:4]}: {data[4:]}"
    print(f"{message}\n")
    return
```

Se imprime el mensaje recibido por pantalla con el tiempo de recepción del mensaje a través de la cabecera pkthdr.

initEthMsg

```
def initEthMsg(interface:str) -> int:

    registerEthCallback(process_ethMsg_frame, ETHTYPE)

    return 0
```

Esta función simplemente registra en el diccionario la función callback para el Ethertype 0x3003

sendEthMsg

```
def sendEthMsg(ip:int, message:bytes) -> bytes:

    # Crear la trama Ethernet
    package = ip.to_bytes(4, 'big') + message # Ip destino + mensaje

    if sendEthernetFrame(package, len(package), ETHTYPE, broadcast) ==
-1: # Enviar la trama Ethernet
        return None

    return len(package)
```

Esta función distribuye un mensaje por la red. Primero crea el paquete con la ip destino y el mensaje y después se ejecuta sendEthernetFrame para distribuirlo.

arp.py

```
ARPETHTYPE = 0x0806 #Tipo Ethernet para ARP
```

Primero se ha añadido una constante para el Ethertype del protocolo ARP

printCache

```
def printCache()->None:
    print('{:>12}\t\t{:>12}'.format('IP', 'MAC'))
    with cacheLock:
        for k in list(cache.keys()):
            try:
                print('{:>12}\t\t{:>12}'.format(
                    socket.inet_ntoa(struct.pack('!I', k)),
                    ':'.join(['{:02X}'.format(b) for b in cache[k]])
                ))
            except KeyError:
```

```
                pass #Si se produce una excepción es que se ha
eliminado un elemento de la caché durante el print y lo ignoramos
```

Se ha modificado la función printCache ya implementada para evitar un error que se producía cuando se imprimía la caché y daba la casualidad de que se producía el timeout de persistencia de las claves en el diccionario. La solución propuesta es ignorar el caso de que se borrase una clave en el momento de imprimir, manteniendo la clave borrada por seguridad.

processARPRequest

```
def processARPRequest(data:bytes,MAC:bytes)->None:

    mac_origen = data[2:8]

    if mac_origen != MAC:
        logging.error(f"MAC origen: {mac_origen} MAC recibida: {MAC}")
        return

    ip_origen = data[8:12]
    ip_destino = data[18:22]

    logging.debug(f"IP origen: {ip_origen} IP destino: {ip_destino} MY
IP: {myIP.to_bytes(4, 'big')}")

    if ip_destino != myIP.to_bytes(4, 'big'): # Si la IP destino no es
la propia
        return

    reply = createARPReply(int.from_bytes(ip_origen, 'big'),
mac_origen) # Construimos respuesta ARP con la MAC origen como destino
    sendEthernetFrame(reply, len(reply), ARPETHTYPE, mac_origen) #
Enviamos respuesta ARP

    return
```

Esta función procesa una request ARP . Si la ip de destino es la ip propia se crea una respuesta dirigida a la ip origen y se distribuye por la red dirigida solo a la MAC que ha realizado el request.

processARPReply

```
def processARPReply(data:bytes,MAC:bytes)->None:

    global requestedIP,resolvedMAC,awaitingResponse,cache
```

```
    mac_origen = data[2:8]

    if mac_origen != MAC:
        logging.error(f"Direcciones MAC distintas: Mac origen:
{mac_origen} MAC recibida: {MAC}")
        return

    ip_origen = data[8:12]
    ip_destino = data[18:22]

    if ip_destino != myIP.to_bytes(4, 'big'):
        logging.error(f"IP destino: {ip_destino} MY IP:
{myIP.to_bytes(4, 'big')}")
        return

    if ip_origen != requestedIP.to_bytes(4, 'big'):
        logging.error(f"Ip distinta a la solicitada: IP origen:
{ip_origen} IP solicitada: {requestedIP}")
        return

    with globalLock: # Accedemos a las variables globales de forma
segura
        resolvedMAC = mac_origen
        with cacheLock:
            cache[int.from_bytes(ip_origen, 'big')] = mac_origen
        awaitingResponse = False # Se asigna awaitingResponse a False
para indicar que ya se enviado la respuesta

    return
```

Esta función procesa la respuesta recibida por el dispositivo al que se le ha solicitado la MAC. Si la ip origen es la misma que la del request resuelve la MAC de esa IP y la añade al diccionario de la caché.

createARPRequest

```
def createARPRequest(ip:int) -> bytes:

    global myMAC,myIP
    frame = bytes()
    frame += ARPHeader
    frame += bytes([0x00, 0x01])  # Opcode para request
    frame += myMAC
```

```
    frame += myIP.to_bytes(4, 'big')
    frame += broadcastAddr
    frame += ip.to_bytes(4, 'big')
    return frame
```

En esta función se crea un ARP request con el opcode a 0x0001 para que se ejecute process_arp_request con la MAC destino como broadcast y la IP destino, la IP de la que se quiere la MAC

createARPReply

```
def createARPReply(IP:int ,MAC:bytes) -> bytes:

    global myMAC,myIP
    frame = bytes()
    frame += ARPHeader
    frame += bytes([0x00, 0x02])  # Opcode para respuesta
    frame += myMAC
    frame += myIP.to_bytes(4, 'big')
    frame += MAC
    frame += IP.to_bytes(4, 'big')
    logging.debug(f"REPLY: {frame}")
    return frame
```

Esta función crea un ARP reply. Es una implementación similar a la anterior con la diferencia de que el opcode es 0x0002 para que se ejecute el process_ARPReply y la MAC destino será la MAC origen del ARPRequest.

process_arp_frame

```
def
process_arp_frame(us:ctypes.c_void_p,header:pcap_pkthdr,data:bytes,srcM
ac:bytes) -> None:

    if len(data) < ARP_HLEN:
        logging.error(f"Trama ARP demasiado corta: {len(data)}")
        return

    arp_header = data[:ARP_HLEN]
    if arp_header != ARPHeader:
        logging.error(f"ARP header incorrecto: {arp_header}")
        return
```

```
    opcode = int.from_bytes(data[6:8], 'big') # Extraemos el opcode
para saber si es una petición o una respuesta

    logging.debug(f"Opcode: {opcode}")

    if opcode == 0x0001:
        processARPRequest(data[ARP_HLEN:], srcMac)
    elif opcode == 0x0002:
        processARPReply(data[ARP_HLEN:], srcMac)

    return
```

Esta función es la función de callback para el Ethertype 0x0806 que se encarga de escoger según el opcode que función ejecutar: si el opcode es 1 se ejecutará processARPRequest y si es 2 se ejecutará el processARPReply.

initARP

```
def initARP(interface:str) -> int:

    global myIP,myMAC,arpInitialized

    if globals().get('arpInitialized', False) != True: # Si el nivel
ARP no está inicializado registramos la función de callback
        registerEthCallback(process_arp_frame,ARPETHTYPE)
        myIP = getIP(interface)
        myMAC = getHwAddr(interface)

    if ARPResolution(myIP) != None: # Realizamos una petición ARP
gratuita para comprobar si la IP propia ya está asignada
        with cacheLock:
            if myIP in cache:
                if cache[myIP] != myMAC: # Si la IP está en caché
compronbamos si la MAC asociada es la nuestra
                    return -1

    with cacheLock: # Añadimos la IP con la MAC asociada a la caché
        cache[myIP] = myMAC

    arpInitialized = True
    return 0
```

Esta función se encarga de inicializar el nivel ARP. Como esta función la vamos a reciclar para implementar el ARP gratuito primero se comprueba si el nivel ARP se ha inicializado

anteriormente y si no se ha inicializado se registra en el diccionario la función de callback además de asignar la IP y la MAC.
Después se hace un ARPResolution con la IP siendo la propia IP (ARP gratuito). Si encuentra alguna coincidencia comprueba que la dirección MAC asociada a esa IP es la nuestra, es decir, nos estaríamos encontrando a nosotros mismos. Si no son iguales significa que nuestra IP está asignada en otro equipo y se devolverá error.

ARPResolution

```
def ARPResolution(ip:int) -> bytes:

    global requestedIP,awaitingResponse,resolvedMAC
    with cacheLock: # Comprobamos si la IP solicitada está en caché
        if ip in cache:
            logging.debug("Ip en cache")
            return cache[ip]

    frame = createARPRequest(ip)

    logging.debug(f"FRAME: {frame}")

    with globalLock: # Accedemos a las variables globales de forma
segura y las inicializamos
        requestedIP = ip
        awaitingResponse = True
        resolvedMAC = None

    for _ in range(3): # Enviamos la petición ARP hasta 3 veces

        if sendEthernetFrame(frame, len(frame), ARPETHTYPE,
broadcastAddr) == -1:
            logging.error("Error al enviar trama ARP")
            return None

        time.sleep(0.5)

        with globalLock:
            if awaitingResponse == False:
                logging.debug("MAC RESUELTA")
                return resolvedMAC

    return None
```

Esta función realiza el protocolo ARP. Primero se comprueba si la IP solicitada está en la caché, si es el caso se devuelve la MAC resuelta, si no se procede a crear la trama. Después se inicializan las variables del Globallock para acceder a ellas de forma segura y se realiza la petición 3 veces, si no se encuentra la IP solicitada. Si se ha recibido una respuesta se devuelve la MAC resuelta.

practica2.py

En este archivo solo se han modificado tres partes:

```
    print ( "\tg : Arp gratuito\n")
```

Se ha añadido un print en la ayuda para que introduciendo g se realice un ARP gratuito.

```
    if initEthMsg(args.interface) != 0:
        logging.error('EthMsg no inicializado')
        stopEthernetLevel()
        sys.exit(-1)
```

Se ha añadido la inicialización del nivel de mensaje.

```
elif comando == 'g':
                try:
                    ret = initARP(args.interface)
                    if ret == -1:
                        print(f"Direccion IP duplicada\n")
                    else:
                        print('Dirección IP no duplicada\n')
                except OSError:
                    print('Formato de IP incorrecta\n')
```

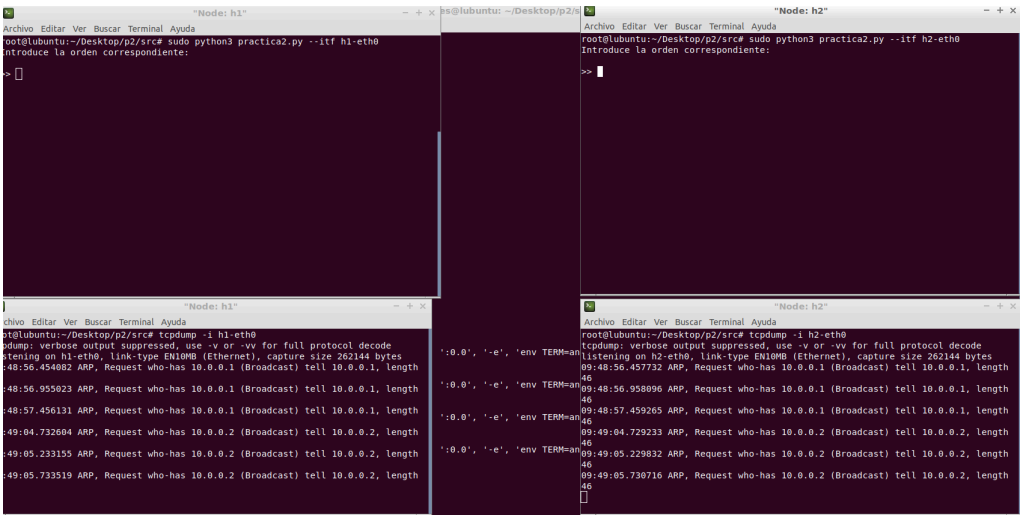
Se ha añadido la funcionalidad en el caso de escribir g (ARP gratuito). Que ejecutará un initARP con la diferencia de que como el nivel ARP ya ha sido inicializado no se asignará una nueva IP ni una MAC, solo realizará el ARPResolution con la propia IP.

Pruebas de ejecución

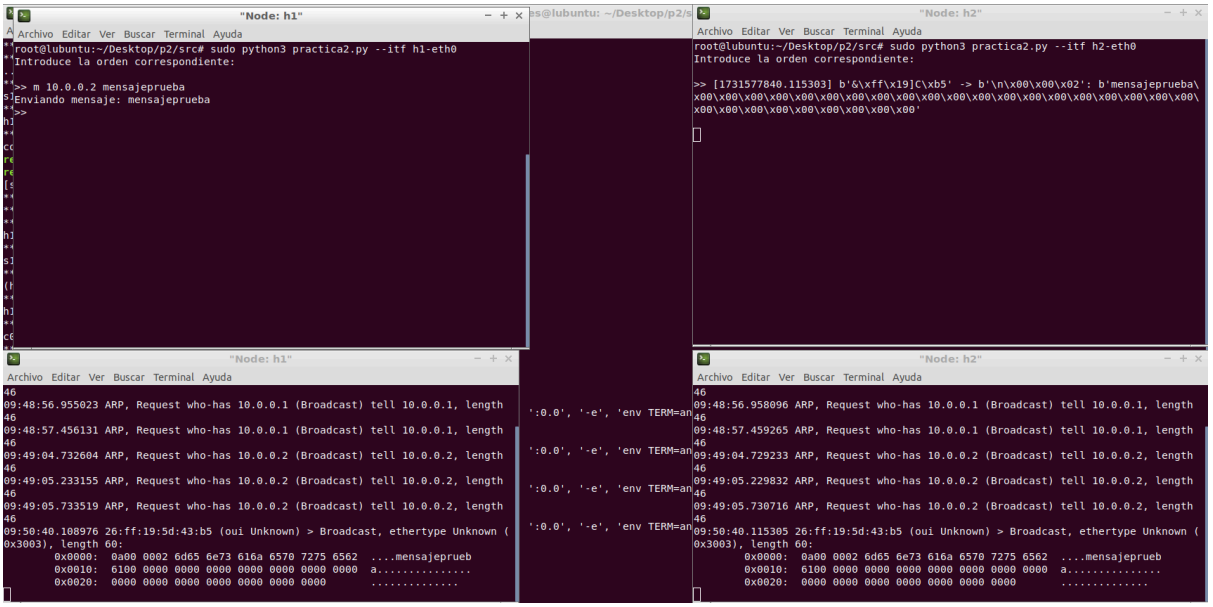
Para comprobar que los paquetes se transfieren y se reciben de forma correcta se ha utilizado este comando de bash por cada interfaz:

```
root@lubuntu:~/Desktop/p2/src# tcpdump -i h1-eth0
```

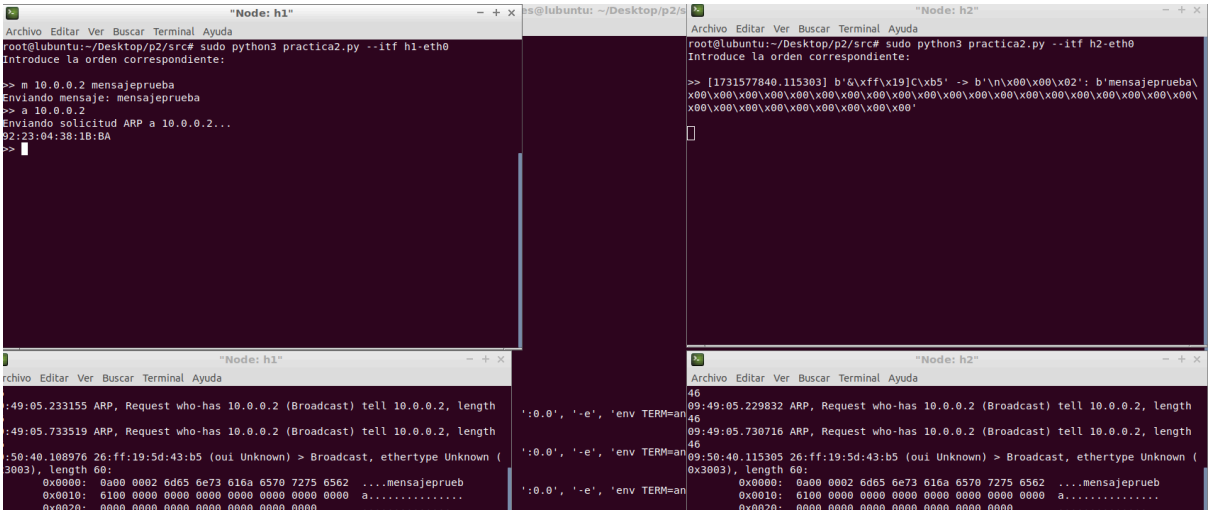
Primero al ejecutar practica2.py en cada una de las interfaces comprobamos que se hace un ARP gratuito en su inicialización:



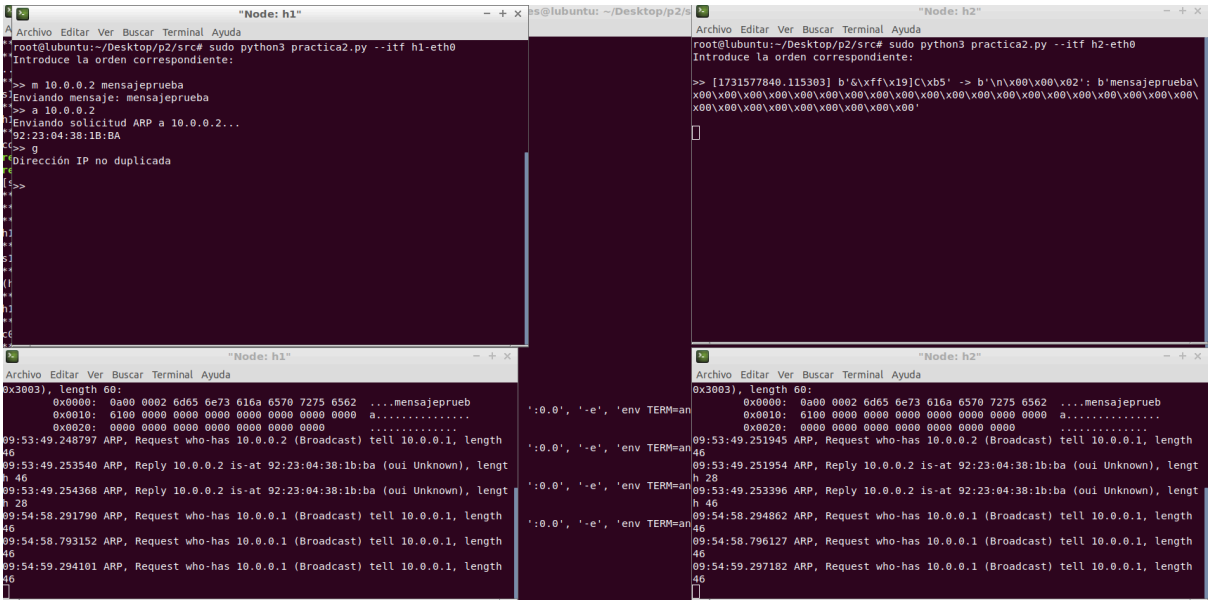
Después comprobamos que enviando un mensaje de prueba por h1, h2 lo recibe:



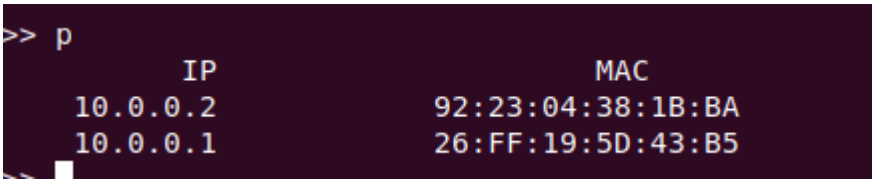
Realizamos un ARP request para saber la MAC de h2 (IP: 10.0.0.2):



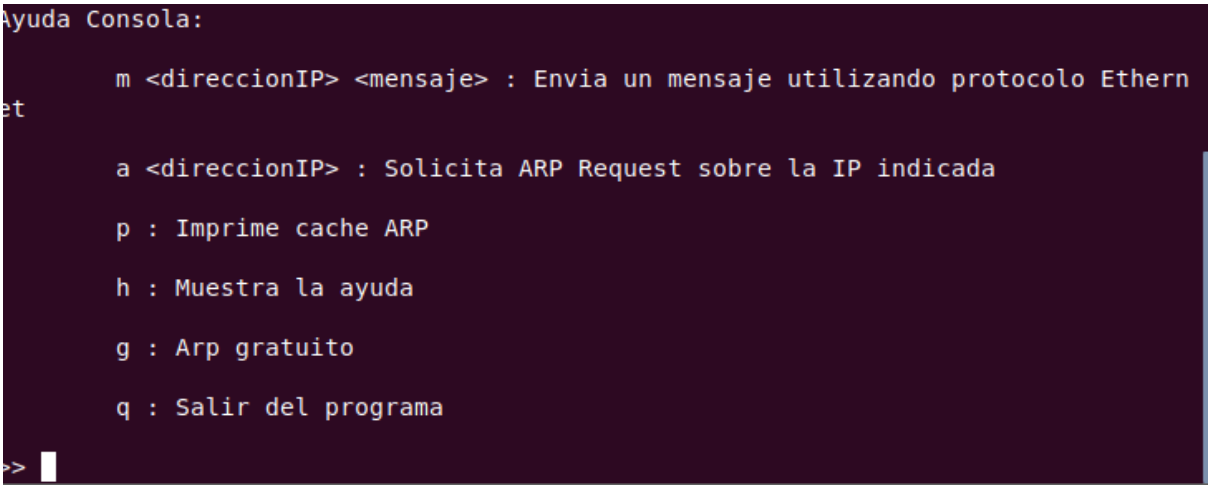
Y si realizamos un ARP gratuito con h1 comprobamos que nuestra dirección IP no está duplicada:



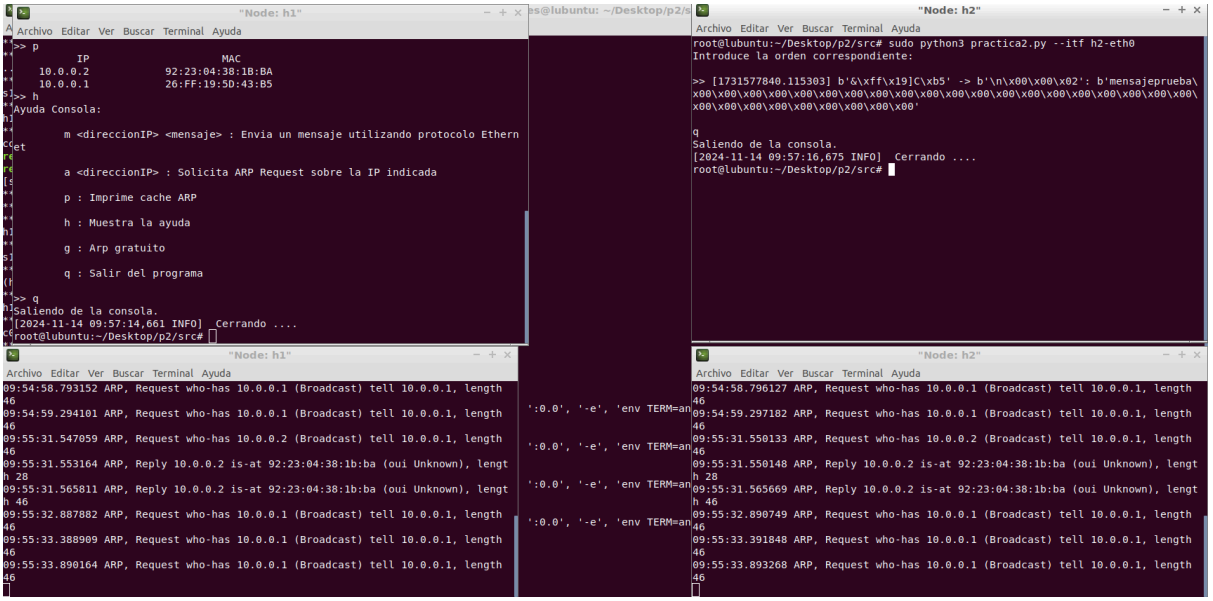
Y si imprimimos la caché comprobamos que tenemos las dos MACs:



Si ejecutamos h podemos ver la ayuda de ejecución:



y si ejecutamos q salimos del programa:



Para ver más detalles de la ejecución se puede usar la opción –debug.