Objetivos de la práctica

- Familiarizarse con el protocolo de nivel de red IP y con los protocolos UDP e ICMP
- Entender las cabeceras y aprender a implementar protocolos de comunicaciones complejos
- Implementar un programa con funcionalidades similares al comando ping.

Recomendación

Se recomienda revisar los siguientes vídeos de la parte de teoría:

- Direcciones IP y MAC https://youtu.be/1ecJdKJPRdE
- Direcciones IP y MAC salto a salto https://youtu.be/6XJn03PI7dg
- Tablas de rutas https://youtu.be/4z6umbTIvTE
- Fragmentación IP https://youtu.be/Hq58yAR89sk
- ICMP https://youtu.be/Cbjg2UmkcXI
- TCP y UDP https://www.youtube.com/watch?v=wk5RJBUm2SA

Introducción

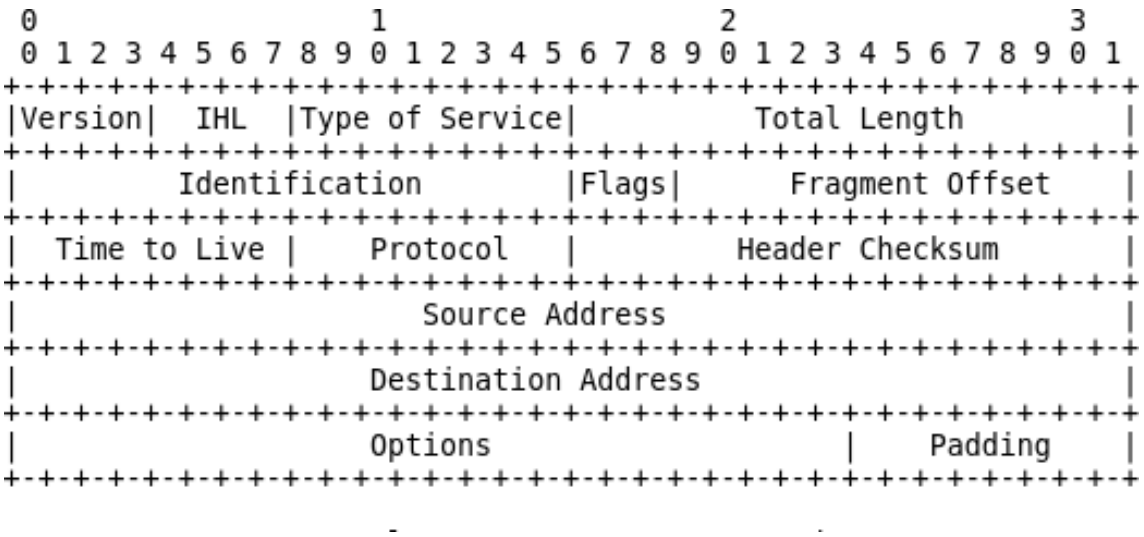
Esta práctica tiene por objeto implementar los niveles intermedios de la pila TCP/IP. Específicamente se construirá el nivel de red que en este caso se implementará usando el protocolo IP y el nivel de transporte que se implementará usando el protocolo UDP. Adicionalmente implementaremos un caso de uso del protocolo ICMP, el cual sirve para notificar errores y realizar tareas de operación relacionadas con el nivel IP.

El objetivo final de esta práctica es implementar un programa que nos permita enviar tanto datagramas UDP que contengan datos útiles como mensajes ICMP de tipo Echo request/reply comúnmente conocidos como "ping".

A continuación, analizaremos las cabeceras de IP, UDP e ICMP. Podemos obtener la descripción de cabeceras pedidas en sus respectivas RFCs (de aquí en castellano: http://www.rfc-es.org/).

Internet Procotol v4(RFC 790):

IP es un protocolo de nivel de red que permite enviar información extremo a extremo a lo largo de varios saltos. A continuación analizaremos el formato que tiene la cabecera de un datagrama IP.



Campos:

- **Versión (4 bits):** Campo que indica la versión de IP. En nuestro caso será siempre 4
- **IHL (4 bits):** Longitud de la cabecera IP. Como la cabecera IP puede contener opciones de tamaño variable este campo nos indica el tamaño de la cabecera. Este campo está expresado en palabras de 4 bytes. Es decir, para obtener el tamaño total (en número de bytes) de la cabecera es necesario multiplicar este campo por 4. El tamaño mínimo de una cabecera IP es 20 bytes (IHL=0x5) y el máximo 60 (IHL=0xF).
- **Type of Service (1 byte):** indicador del tipo de tráfico que transporta este datagrama. Este campo sirve para priorización y marcado de tráfico. En nuestro caso siempre usaremos el valor 1.
- **Total Length(2 Bytes):** Longitud total (en número de bytes) del datagrama IP actual. Incluye tanto la cabecera como el payload que va detrás de la cabecera.
- **Identification (2 Bytes):** Identificador del datagrama IP (también llamado IPID). Este campo es útil cuando hay fragmentación IP. En este caso todos los fragmentos tienen el mismo valor de IPID. Para los envíos, este valor se fija inicialmente al arrancar el nivel IP. En la práctica **lo fijaremos al número de pareja** (es necesario modificar el código).

- **Flags (3 bits):** Banderas IP, de izquierda a derecha:
  - **Bit 1 (Reservado):** siempre a 0.

    - **Bit 2 (DF):** bandera que indica que no debe fragmentarse el datagrama. En nuestro caso será siempre 0.
    - **Bit 3 (MF):** bandera que indica que vienen más fragmentos tras el datagrama actual. En caso de fragmentar todos los fragmentos tendrán este bit a 1 menos el último fragmento.
- **Offset (13 bits):** campo que indica (en caso de fragmentación) el *offset* de los datos contenidos en el datagrama actual respecto al total de datos sin fragmentar. Está expresado en palabras de 8 bytes. Es decir, para obtener el valor real de offset se debe multiplicar este campo por 8.
- **Time to Live (1 Byte):** campo que indica el número máximo de saltos IP que puede realizar el datagrama actual antes de ser descartado. Cada vez que un paquete atraviesa un salto a nivel IP se decrementa en 1 y cuando llega a 0 el datagrama actual se descarta. En nuestro caso usaremos siempre el valor por defecto 65.
- **Protocol (1 Byte):** Campo que indica el protocolo de nivel superior encapsulado en el payload del datagrama. Este campo tiene un cometido similar al campo Ethertype en Ethernet. Algunos valores típicos son: 1 para ICMP, 6 para TCP y 17 para UDP.
- **Header Checksum (2 Bytes):** suma de verificación calculada sobre la cabecera IP que sirve para detectar errores o modificaciones de la cabecera IP durante el envío de datos. Cuando recibimos un datagrama, si el cálculo de checksum es erróneo debemos descartarlo.
- **Dirección IP origen (4 Bytes):** dirección IP del emisor del datagrama actual
- **Dirección IP destino (4 Bytes):** dirección IP del receptor del datagrama
- **Opciones (Tamaño variable):** Opciones que aportan funcionalidades adicionales. Su tamaño tiene que ser múltiplo 4 bytes. El tamaño mínimo de opciones es 0 bytes y el máximo 40 bytes.

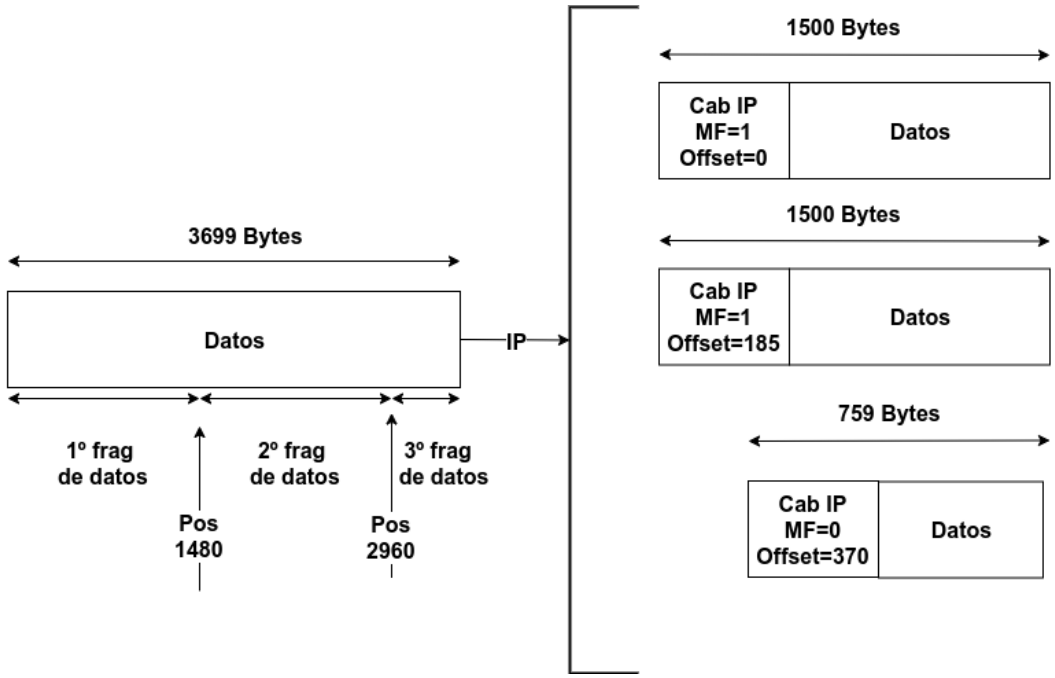
Tamaños de los datagramas:

- El tamaño máximo que puede enviarse a nivel IP (cabecera + datos) viene definido por la *Maximum Transmission Unit* (MTU). Este valor depende del protocolo de nivel inferior, en nuestro caso Ethernet. En nuestro caso la MTU es típicamente 1500 Bytes **pero este valor puede variarse mediante el comando ifconfig.**
- Cuando los datos a enviar no caben en un único datagrama IP se realiza un proceso de fragmentación. Para ello se parten los datos en fragmentos y cada fragmento se envía junto con su propia cabecera IP. En este caso cada fragmento tendrá un valor diferente en los campos Total Length, MF, offset y checksum. El resto de campos serán los mismos en todos los fragmentos.

Ejemplo de fragmentación:

1. Queremos enviar 3699 bytes de datos utilizando el protocolo IP.
2. Sabiendo que la MTU es 1500 primeramente debemos calcular el número de fragmentos a enviar. Para ello calculamos primeramente la cantidad máxima de datos útiles que podemos encapsular en un datagrama IP. Esta cantidad será: MTU-long_cabecera_IP. Si suponemos que la cabecera no tiene opciones este valor sería 1500-20=1480 Bytes.
3. A continuación debemos comprobar si esta cantidad máxima de datos útiles es múltiplo de 8. Esta comprobación es obligatoria porque el offset está expresado en palabras de 8 bytes. En caso de que no lo sea usaremos el valor más cercano por abajo. Como 1480 sí es múltiplo de 8 usaremos ese valor.
4. Calculamos el número de fragmentos a enviar. Para ello dividimos el total de datos a enviar/cantidad máxima de datos útiles. En nuestro ejemplo: 3699/1480=2,49. Como no podemos enviar un número decimal de fragmentos enviaremos 3 fragmentos (el último no irá completamente lleno).
5. El primer fragmento tendrá longitud 20 (cabecera) + 1480 (datos) bytes . El valor de MF será 1 (hay más fragmentos detrás) y el offset será 0 (los 1480 bytes contenidos en este primer datagrama empiezan en la posición 0 respecto al inicio de los datos originales).
6. El segundo fragmento tendrá longitud 20 (cabecera) + 1480 (datos) bytes . El valor de MF será 1 (hay más fragmentos detrás) y el offset será 1480/8=185 (los datos contenidos en este segundo datagrama empiezan en la posición 1480 respecto al inicio de los datos originales).
7. El ultimo fragmento tendrá longitud 20 (cabecera) + 739 (datos) bytes. El valor de MF=0 (este es el último fragmento) y el offset será 2960/8=370 (los datos contenidos en este último datagrama empiezan en la posición 2960 respecto al inicio de los datos originales)

En la siguiente imagen se muestra este ejemplo:



Cambio de MTU de una interfaz:

Para hacer pruebas de fragmentación se recomienda cambiar la MTU de la interfaz. Para cambiar la MTU de una interfaz en Linux debemos ejecutar en una terminal:

ifconfig nombre_interfaz mtu valor_mtu

Ejemplo:

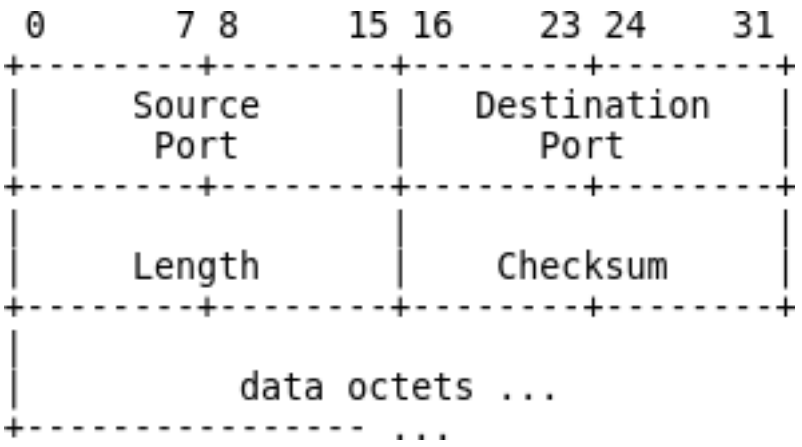
ifconfig h1-eth0 mtu 600

UDP(RFC 768):

UDP es un protocolo de nivel de transporte no fiable y no orientado a conexión que permite enviar datagramas. UDP se usa normalmente para implementar por encima protocolos de nivel de aplicación no orientados a flujo o que tienen restricciones de tiempo real y en los cuales el concepto de retransmisión no tiene sentido. Por ejemplo los protocolos de nivel de aplicación DNS, DHCP o RTP son protocolos que hacen uso de UDP como nivel de transporte.

Cabecera:

El protocolo UDP tiene una cabecera fija de 8 bytes. A continuación se muestra dicha cabecera:



Campos:

- **Source Port (2 Bytes):** Indica el puerto origen del datagrama UDP.
- **Destination Port (2 Bytes):** Indica el puerto destino del datagrama UDP.
- **Length (2 Bytes):** Tamaño (en bytes) del datagrama UDP contando la cabecera y los datos.
- **Checksum (2 bytes):** Suma de comprobación sobre la pseudocabecera IP. El uso del campo es opcional y en la práctica usaremos siempre el valor 0.

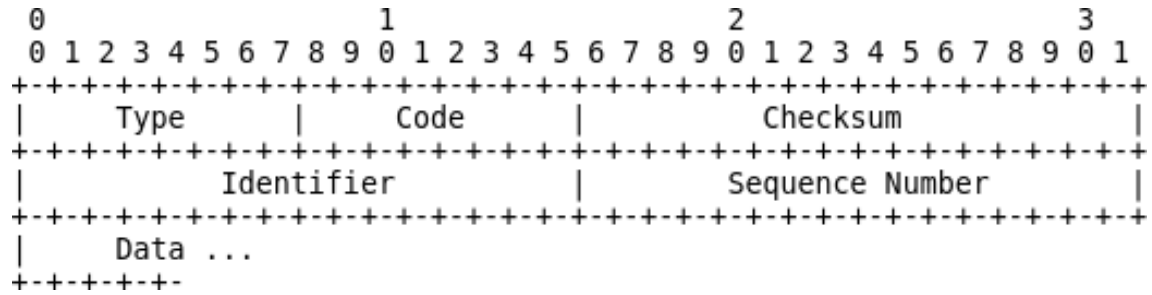
ICMP(RFC 792):

ICMP es un protocolo de control, diagnóstico y notificación de errores relacionado estrechamente con IP. Aunque funciona por encima de IP no es un protocolo de transporte ya que no se usa para intercambiar información extremo a extremo entre sistemas o aplicaciones.

Cabecera:

La cabecera ICMP tiene una parte fija (primeros 32 bits) y común para todos los tipos de mensaje y una parte variable que depende del tipo de mensaje o notificación. Para el desarrollo de las prácticas vamos a centrarnos en los mensajes **ICMP Echo Request** e **ICMP Echo Reply** que se usan típicamente para implementar la aplicación ping. Esta aplicación se usa comúnmente para comprobar la conectividad a nivel IP con equipos y para estimar su *Round-Trip Time* (RTT).

A continuación se muestra el formato de la cabecera para mensajes de tipo Echo Request y Echo Reply:



- **Type (1 Byte):** Indica el tipo de mensaje ICMP. En el caso de los mensajes Echo Request este campo vale 8 mientras que en el caso de los mensajes Echo Reply vale 0.
- **Code (1 Byte):** Indica el subtipo de mensaje ICMP. Tanto para Echo Request como para Echo Reply este valor es 0.
- **Checksum (2 Bytes):** Suma de comprobación (similar a la de IP). Incluye tanto la cabecera como los datos. Si la longitud del mensaje (cabecera + datos) es impar se debe añadir un byte a 0 en los datos para convertirlo en par.
- **Identifier (2 Bytes):** Identificador numérico que se usa para ayudar a correlar Echo Request con sus correspondientes Echo Reply. Típicamente se suele elegir como identificador el PID del proceso que genera los Echo Request. En nuestro caso **usaremos siempre el número de pareja**. Este identificador no cambia a lo largo de la vida del proceso.
- **Sequence Number (2 Bytes):** Número de secuencia que ayuda a correlar preguntas y respuestas. Este identificador se transmite en orden de red e incrementa (típicamente en 1) con cada mensaje Echo Request que se envía. El número de secuencia comenzará en 0.

Funcionamiento del comando ping:

El comando ping se usa para comprobar la conectividad a nivel IP con equipos y para estimar su *Round-Trip Time* (RTT). Este comando está presente en casi todos los sistemas operativos modernos. El funcionamiento general de este comando es el que sigue:

1. Un host (por ejemplo h1) envía un mensaje ICMP Echo Request a otro host (por ejemplo h2) con un Identifier y un Sequence Number dado (por ejemplo 0 y 0)
2. Se obtiene y se guarda el timestamp (tiempo) de envío del paquete Echo Request anteriormente enviado
3. Típicamente el host destino enviará un Echo Reply con los mismos valores de Identifier y Sequence Number (en nuestro caso 0 y 0) como respuesta

4. Cuando este mensaje llega al host original se guarda su tiempo de llegada
5. Para obtener el RTT se resta el tiempo de llegada menos el de envío (comprobando que los valores de Identifier y Sequence Number son los mismos entre el Echo Request enviado y Echo Reply recibido).
6. Este proceso puede repetirse más veces para obtener más muestras de RTT incrementando el valor de Sequence Number pero dejando fijo el Identifier

Ejercicios

El objetivo final de esta práctica es implementar un programa que envíe tanto datagramas UDP como mensajes de tipo ping. Para poder realizar estas acciones será necesario implementar IP, UDP e ICMP. En este sentido se propone partir del código que se proporciona, completar y ampliar el código entregado en los ficheros ip.py, udp.py e icmp.py. Además de completar estos ficheros será necesario usar los niveles Ethernet y ARP implementados en la anterior práctica para construir sobre ellos IP (ficheros ethernet.py y arp.py)

Adicionalmente, el programa deberá disponer de la opción --icmpsize <valor>, en este caso el programa mandara paquetes ICMP request del tamaño en bytes indicado (<valor>), rellenando en este caso el campo datos del paquete ICMP con caracteres de la "A" a la "Z" de manera consecutiva hasta completar el tamaño. Esta opción no esta incluida en el material de la practica y deberá ser complementada por el alumno.

A continuación se describen las funciones que deben ser implementadas en los ficheros Python:

ip.py:

Funciones ya implementadas:

chksum(msg)

Descripción:

- Esta función calcula el checksum IP sobre unos datos de entrada dados (msg)

Argumentos:

- **msg:** array de bytes con el contenido sobre el que se calculará el checksum

Retorno:

- Entero de 16 bits con el resultado del checksum en **ORDEN DE RED (no debe usar ! al llamar a struct.pack)**

getMTU(interface)

Descripción:

- Esta función obtiene la MTU para un interfaz dada

Argumentos:

- **interface:** cadena con el nombre la interfaz sobre la que consultar la MTU

Retorno:

- Entero con el valor de la MTU para la interfaz especificada

getNetmask(interface)

Descripción:

- Esta función obtiene la máscara de red asignada a una interfaz

Argumentos:

- **interface:** cadena con el nombre la interfaz sobre la que consultar la máscara

Retorno:

- Entero de 32 bits con el valor de la máscara de red

getDefaultGW(interface)

Descripción:

- Esta función obtiene el gateway por defecto para una interfaz dada

Argumentos:

- **interface:** cadena con el nombre la interfaz sobre la que consultar el gateway

Retorno:

- Entero de 32 bits con la IP del gateway

Funciones a implementar:

process_IP_datagram(us,header,data,srcMac)

Descripción:

- Esta función procesa datagramas IP recibidos. Se ejecuta una vez por cada trama Ethernet recibida con Ethertype 0x0800
- Esta función debe realizar, al menos, las siguientes tareas:
    - Extraer los campos de la cabecera IP (incluida la longitud de la cabecera)
    - Calcular el checksum sobre la cabecera y comprobar que el valor es correcto
    - Analizar los bits de MF y el offset. Si el offset tiene un valor != 0 dejar de procesar el datagrama (no vamos a reensamblar)
    - "Loggear" (usando logging.debug) el valor de los siguientes campos:
        - Longitud de la cabecera IP (multiplicada por 4)
        - ToS
        - IPID
        - Valor de las banderas DF y MF
        - Valor decimal de offset (multiplicado por 8)
        - TTL
        - Protocolo
        - IP origen y destino en notación decimal (a.b.c.d)
    - Comprobar si tenemos registrada una función de *callback* de nivel superior consultando el diccionario protocols y usando como clave el valor del campo protocolo del datagrama IP.
    - En caso de que haya una función de nivel superior registrada, debe llamarse a dicha función pasando los datos (*payload*) contenidos en el datagrama IP.

Argumentos:

- **us:** Datos de usuario pasados desde la llamada de pcap_loop. En nuestro caso será None
- **header:** cabecera pcap_pkthdr
- **data:** array de bytes con el contenido del datagrama IP
- **srcMac:** MAC origen de la trama Ethernet que se ha recibido

Retorno:

- Ninguno

registerIPProtocol(callback,protocol)

Descripción:

- Esta función recibirá el nombre de una función y su valor de protocolo IP asociado y añadirá en la tabla (diccionario) de protocolos de nivel superior dicha asociación.
- Este mecanismo nos permite saber a qué función de nivel superior debemos llamar al recibir un datagrama IP con un determinado valor del campo protocolo (por ejemplo TCP o UDP). Por ejemplo, podemos registrar una función llamada process_UDP_datagram asociada al valor de protocolo 17 y otra llamada process_ICMP_message asociada al valor de protocolo 1.

Argumentos:

- **callback**: función de callback a ejecutar cuando se reciba un datagrama con el protocolo especificado como segundo argumento. La función que se pase como argumento debe tener el siguiente prototipo: funcion(us,header,data,srcIp):
    - Dónde:
        - **us:** son los datos de usuario pasados por pcap_loop (en nuestro caso este valor será siempre None
        - **header:** estructura pcap_pkthdr que contiene los campos len, caplen y ts.
        - **data:** payload del datagrama IP. Es decir, la cabecera IP **NUNCA** se pasa hacia arriba.
        - **srcIP:** dirección IP que ha enviado el datagrama actual.
    - La función no retornará nada. Si un datagrama se quiere descartar basta con hacer un *return* sin valor y dejará de procesarse.
- **protocol:** valor del campo protocolo de IP para el cuál se quiere registrar una función de callback.

Retorno:

- Ninguno

initIP(interface,opts=None)

Descripción:

- Esta función inicializará el nivel IP. Esta función debe realizar, al menos, las siguientes tareas:
    - Llamar a initARP para inicializar el nivel ARP realizado en la práctica anterior.
    - Obtener (llamando a las funciones correspondientes) y almacenar en variables globales los siguientes datos:
        - IP propia
        - MTU
        - Máscara de red (netmask)
        - Gateway por defecto
    - Almacenar el valor de opts en la variable global ipOpts
    - Registrar a nivel Ethernet (llamando a registerCallback) la función process_IP_datagram con el Ethertype 0x0800
    - Inicializar el valor de IPID según lo indicado más arriba.

Argumentos:

- **interface:** cadena de texto con el nombre de la interfaz sobre la que inicializar IP
- **opts:** array de bytes con las opciones a nivel IP a incluir en los datagramas o None si no hay opciones a añadir

Retorno:

- True o False en función de si se ha inicializado el nivel o no.

sendIPDatagram(dstIP,data,protocol)

Descripción:

- Esta función construye un datagrama IP y lo envía. En caso de que los datos a enviar sean muy grandes la función debe generar y enviar el número de fragmentos IP que sean necesarios.
- Esta función debe realizar, al menos, las siguientes tareas:
    - Determinar si se debe fragmentar o no y calcular el número de fragmentos a generar

- Para cada datagrama o fragmento:
    - Construir la cabecera IP con los valores que corresponda. Incluir opciones en caso de que ipOpts sea distinto de None
    - Calcular el *checksum* sobre la cabecera y añadirlo
    - Añadir los datos recibidos a la cabecera IP
    - En el caso de que sea un fragmento ajustar los valores de los campos MF y *offset* de manera adecuada
- Enviar el datagrama o fragmento llamando a sendEthernetFrame. Para determinar la dirección MAC de destino al enviar los datagramas usaremos la máscara de red y las IPs.
- Una vez enviado el datagrama o los fragmentos:
    - Incrementar la variable IPID en 1.

Argumentos:

- **dstIP:** entero de 32 bits con la IP destino del datagrama
- **data:** array de bytes con los datos a incluir como payload en el datagrama
- **protocol:** valor numérico del campo IP protocolo que indica el protocolo de nivel superior de los datos contenidos en el payload. Por ejemplo 1, 6 o 17.

Retorno:

- True o False en función de si se ha enviado el datagrama correctamente o no

udp.py:

Funciones ya implementadas:

getUDPSourcePort()

Descripción:

- Esta función obtiene un puerto origen libre en la máquina actual.

Argumentos:

- Ninguno

Retorno:

- Entero de 16 bits con el número de puerto origen disponible.

Funciones a implementar:

process_UDP_datagram(us,header,data,srcIP)

Descripción

- Esta función procesa un datagrama UDP. Esta función se ejecutará por cada datagrama IP que contenga un 17 en el campo protocolo de IP
- Esta función debe realizar, al menos, las siguientes tareas:
    - Extraer los campos de la cabecera UDP
    - "Loggear" (usando logging.debug) los siguientes campos:
        - Puerto origen
        - Puerto destino
        - Datos contenidos en el datagrama UDP

Argumentos:

- **us:** son los datos de usuarios pasados por pcap_loop (en nuestro caso este valor será siempre None)
- **header:** estructura pcap_pkthdr que contiene los campos len, caplen y ts
- **data:** array de bytes con el contenido del datagrama UDP
- **srcIP:** dirección IP que ha enviado el datagrama actual

Retorno:

- Ninguno

sendUDPDatagram(data,dstPort,dstIP)

Descripción:

- Esta función construye un datagrama UDP y lo envía.

Esta función debe realizar, al menos, las siguientes tareas:

- Construir la cabecera UDP:
  - El puerto origen lo obtendremos llamando a getUDPSourcePort
  - El valor de checksum lo pondremos siempre a 0
- Añadir los datos
- Enviar el datagrama resultante llamando a sendIPDatagram

Argumentos:

- **data:** array de bytes con los datos a incluir como payload en el datagrama UDP
- **dstPort:** entero de 16 bits que indica el número de puerto destino a usar
- **dstIP:** entero de 32 bits con la IP destino del datagrama UDP

Retorno:

- True o False en función de si se ha enviado el datagrama correctamente o no

initUDP()

Descripción:

- Esta función inicializa el nivel UDP. La función debe realizar, al menos, las siguientes tareas:
  - Registrar (llamando a registerIPProtocol) la función process_UDP_datagram con el valor de protocolo 17

Argumentos:

- Ninguno

Retorno:

- Ninguno

icmp.py

Funciones a implementar:

process_ICMP_message(us,header,data,srcIP)

Descripción:

- Esta función procesa un mensaje ICMP. Esta función se ejecutará por cada datagrama IP que contenga un 1 en el campo protocolo de IP. Esta función debe realizar, al menos, las siguientes tareas:
    - Calcular el checksum de ICMP y comprobar que es correcto
    - Extraer campos tipo y código de la cabecera ICMP
    - "Loggear" (con logging.debug) el valor de tipo y código
    - Si el tipo es ICMP_ECHO_REQUEST_TYPE:
        - Generar un mensaje de tipo ICMP_ECHO_REPLY como respuesta:
            - Este mensaje debe contener los datos recibidos en el ECHO_REQUEST. Es decir, "rebotamos" los datos que nos llegan.
            - Los valores de icmp_id e icmp_seqnum deben ser los contenidos en el mensaje ECHO_REQUEST.
            - Enviar el mensaje usando la función sendICMPMessage
    - Si el tipo es ICMP_ECHO_REPLY_TYPE:
        - Extraer del diccionario icmp_send_times el valor de tiempo de envío usando como clave los campos srcIP e icmp_id e icmp_seqnum contenidos en el mensaje ICMP. Restar el tiempo de envió extraído con el tiempo de recepción (contenido en la estructura pcap_pkthdr)
        - Se debe proteger el acceso al diccionario de tiempos usando la variable timeLock
        - Mostrar por pantalla la resta. Este valor será una estimación del RTT

- Si es otro tipo:
    - No hacer nada

Argumentos:

- **us:** son los datos de usuarios pasados por pcap_loop (en nuestro caso este valor será siempre None)
- **header:** estructura pcap_pkthdr que contiene los campos len, caplen y ts.
- **data:** array de bytes con el conenido del mensaje ICMP
- **srcIP:** dirección IP que ha enviado el datagrama actual.

Retorno:

- Ninguno

sendICMPMessage(data,type,code,icmp_id,icmp_seqnum,dstIP)

Descripción:

- Esta función construye un mensaje ICMP y lo envía.

Esta función debe realizar, al menos, las siguientes tareas:

- Si el campo type es ICMP_ECHO_REQUEST_TYPE o ICMP_ECHO_REPLY_TYPE:
    - Construir la cabecera ICMP
    - Añadir los datos al mensaje ICMP
    - Calcular el checksum y añadirlo al mensaje donde corresponda
    - Si type es ICMP_ECHO_REQUEST_TYPE
        - Guardar el tiempo de envío (llamando a time.time()) en el diccionario icmp_send_times usando como clave el valor de dstIP+icmp_id+icmp_seqnum
        - Se debe proteger al acceso al diccionario usando la variable timeLock
    - Llamar a sendIPDatagram para enviar el mensaje ICMP
- Si no:
    - Tipo no soportado. Se devuelve False

Argumentos:

- **data:** array de bytes con los datos a incluir como payload en el mensaje ICMP

- **type**: valor del campo tipo de ICMP

- **dstIP**: entero de 32 bits con la IP destino del mensaje ICMP
- **code:** valor del campo code de ICMP
- **icmp_id:** valor del campo ID de ICMP
- **icmp_seqnum:** valor del campo Seqnum de ICMP

initICMP()

Descripción:

- Esta función inicializa el nivel ICMP. La función debe realizar, al menos, la siguiente tarea:
    - Registrar (llamando a registerIPProtocol) la función process_ICMP_message con el valor de protocolo 1

Argumentos:

- Ninguno

Retorno:

- Ninguno

Ejecución de pruebas:

Para realizar las pruebas durante el desarrollo y la validación final se hará uso de la herramienta Mininet. Se puede revisar el funcionamiento de dicha herramienta en el documento de entorno de trabajo al inicio de la sección de prácticas. Ejecutaremos al menos 4 tipos de pruebas:

1. Pruebas de envío de datagramas UDP:

1. Partiremos de la configuración básica en la que hay 2 hosts (h1 y h2).
2. Ejecutaremos nuestro programa en h1 y h2 con la bandera --debug y con la IP destino de h2 y h1 respectivamente.
3. En una nueva terminal de h2 ejecutaremos Wireshark
4. Desde h1 enviaremos un datagrama UDP
5. Comprobaremos en Wireshark que el datagrama UDP (y el IP sobre el que está transportado) son correctos
6. Comprobaremos en h2 que la salida de logging.debug muestra por pantalla los campos solicitados
7. Repetiremos la misma prueba con la opción del programa --addOptions
8. Repetiremos la prueba original con la opción --dataFile pasando un archivo de más de 3000 caracteres
9. Repetiremos la prueba anterior cambiando la MTU de la interfaz a un valor más bajo (por ejemplo 658)

2. Pruebas de envío de ICMP:

1. Partiremos de la configuración básica en la que hay dos hosts (h1 y h2).
2. Ejecutaremos nuestro programa en h1 y h2 con la bandera --debug y con la IP destino de h2 y h1 respectivamente.
3. En una nueva terminal de h2 ejecutaremos Wireshark
4. Desde h1 enviaremos un mensaje de tipo Echo Request a h2
5. Comprobaremos en Wireshark que tanto el mensaje ICMP Echo Request como el ICMP Echo Reply generados son correctos. Si todo es correcto deberemos observar 2 mensajes ICMP Echo Reply. Uno corresponde al generado por nuestro programa y otro corresponde al generado por el sistema operativo.
6. Comprobaremos en h2 que la salida de logging.debug muestra por pantalla los campos solicitados
7. Comprobaremos en h que se muestra por pantalla la estimación del RTT
8. Repetiremos la misma prueba con la opción del programa --addOptions

3. Pruebas de envío de datagramas UDP a hosts fuera de la subred:

1. Partiremos de la configuración básica en la que hay dos hosts (h1 y h2).

2. Ejecutaremos nuestro programa en h1 usando como dirección IP destino una dirección de fuera de la subred (por ejemplo 8.8.8.8)
3. En una nueva terminal de h1 ejecutaremos Wireshark
4. Realizaremos el envío del datagrama UDP
5. Analizaremos en Wireshark que la dirección MAC destino presente en la trama Ethernet se corresponde con la MAC del gateway (10.0.0.3)

4. Pruebas de compatibilidad con ping estándar de Linux:

1. Partiremos de la configuración básica en la que hay dos hosts (h1 y h2).
2. Ejecutaremos nuestro programa en h1 usando como dirección IP cualquiera (por ejemplo 8.8.8.8 pues va a ser irrelevante)
3. En una nueva terminal de h1 ejecutaremos Wireshark
4. Desde una terminal en h2 ejecutaremos ping 10.0.0.1
5. Analizaremos en h2 que la salida del comando ping indica que hay respuestas duplicadas. Si esto no ocurre entonces el programa no será correcto ni compatible. A continuación se muestra un ejemplo de salida donde esto ocurre:

64 bytes from 10.0.0.1: icmp_seq=0 ttl=64 time=220 ms (DUP!)

Para todas las pruebas se recomienda el uso de Wireshark para validar la correcta construcción y envío de datos.

Operaciones a nivel de bit

En Python, la unidad mínima de información que se puede declarar de manera estándar es un byte. En muchas ocasiones los protocolos que analizaremos o construiremos usarán campos a nivel de bit. Para ello debemos saber manejar este tipo de situaciones. En Python existen los siguientes operadores binarios:

- **| (or lógico)**
- **& (and lógico)**
- **<< (desplazamiento a la izquierda)**
- **>> (desplazamiento a la derecha)**
- **^ (xor lógico)**
- **~ (Complemento a 1)**

Haciendo uso de estos operadores podemos extraer o construir campos a nivel de bit. A continuación se muestra un ejemplo de cómo poner a 1 el bit 12 de una variable de 16 bits:

dato=0x01B0
uno=1

dato=dato|(uno << 12)

Si imprimimos la variable dato en hexadecimal obtendremos el valor 0x11B0. Para obtener esto hemos hecho un desplazamiento a la izquierda del número uno en 12 posiciones de tal manera que hemos obtenido una variable que tiene a 0 todos los bits menos el bit 12. Luego se ha realizado un or lógico con la variable que deseamos modificar. De esta manera si en el bit 12 de la variable dato había un 1 quedará un 1 y si había un 0 quedará igualmente un 1.

Con este ejemplo hemos visto como cambiar el valor de un bit. Si lo que queremos es saber el valor de un bit determinado realizaremos un proceso similar. En el siguiente ejemplo se muestra como sacar el valor del bit 12 de una variable.

dato=0x11B0
valor=0

valor=(dato&0x1000)>>12

Si imprimimos la variable valor obtendremos el valor 1 ya que el bit 12 de la variable dato es un 1.Para obtener este valor hemos realizado un and lógico de la variable de la que queremos extraer el valor con una máscara. Dicha máscara es un número de 16 bits que tiene todos sus bits a 0 menos el correspondiente a la posición 12 (que es la que queremos extraer). De esta manera, después de realizar la operación and tendremos el valor del bit 12 almacenado en una variable de 16 bits. Como nos interesa tener un único bit, (0 ó 1), debemos mover hacia la derecha con el operador >> 12 posiciones.

Criterios de evaluación

Ejercicios: Entrega especificada en la actividad de Moodle

- Normativa de entrega cumplida en su totalidad: 2,5%

- Fichero leeme.txt bien explicado: 2,5%
- Recibir y enviar datagramas UDP sin opciones IP ni fragmentación 25%
- Recibir y enviar mensajes ICMP sin opciones IP ni fragmentación (incluyendo pruebas con ping estándar) 20%
- Enviar y recibir correctamente mensajes ICMP con opciones IP 10%
- Enviar y recibir mensajes ICMP con un tamaño determinado 5%
- Enviar datagramas UDP con fragmentación 10%
- Enviar datagramas UDP con fragmentación y opciones IP 10%
- Enviar datagramas UDP fuera de la subred actual 15%

Control individual: Cuestionario final sobre la práctica el día especificado en Moodle. No olvide ser puntual, el control empezará a "y 10".

Entrega

Archivos fuente completos **practica3.py, rc1_pcap.py, ethernet.py , arp.py, ip.py, udp.py e icmp.py**.

- Añada un archivo **leeme.txt** que incluya los nombres de los autores, comentarios que se quieran transmitir al profesor y, en caso de entregar algún archivo más, la descripción y/o explicación del mismo. **Además este fichero debe contener una sección donde se determine si se ha dado respuesta (Realizado/Parcialmente-Realizado/No-Realizado, y en caso afirmativo la explicación de cómo se ha validado) a cada criterio de evaluación solicitado.**

Comprima en un zip **TODO** lo que vaya a entregar y llámelo practica3_YYYY_PXX.zip, donde YYYY es el grupo al que pertenece (miercoles1 miercoles2, viernes1, viernes2, viernes3), y XX (y solo XX) es el número de pareja (**con dos dígitos**).

Por ejemplo, para la pareja 5 del grupo viernes4: **$ zip practica3_viernes4_P05.zip ***

Solo es necesario que suba la entrega un miembro de la pareja.