PRÁCTICA 3: Cliente-servidor UDP

El objetivo de esta práctica es que te familiarices con el desarrollo de aplicaciones sencillas que usan los servicios de transporte del protocolo UDP. A lo largo de la práctica se realizarán diversas pruebas con una aplicación distribuida escrita en lenguaje Python que sigue la arquitectura cliente/servidor. El servicio que realiza dicha aplicación consiste en hacer eco; es decir, el servidor recibe un mensaje y lo reenvía de nuevo al cliente que se lo envió. Es preferible que realices esta aplicación en Linux.

Recuerda que cuando la comunicación entre dos procesos de aplicación usa los servicios de transporte UDP, cada extremo de la comunicación debe crear un socket UDP. El proceso que realiza el lado servidor debe asociar su socket a una dirección local (IP:puerto) conocida por el cliente. Quizás quieras repasar la práctica anterior antes de continuar para recordar cómo se direccionan los procesos de aplicación y sus sockets asociados.

INTRODUCCIÓN AL USO DE SOCKETS UDP

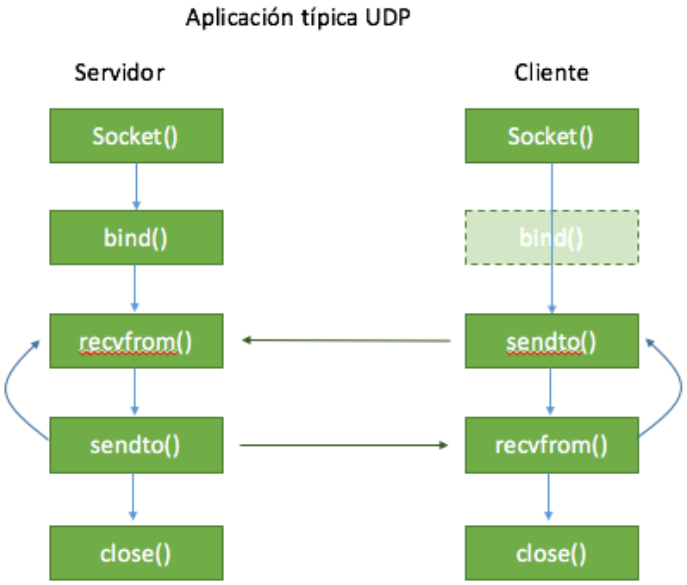
En la práctica anterior aprendiste cómo crear un socket de tipo UDP y cómo asignarle una dirección local si fuera necesario. En la siguiente tabla puedes encontrar un resumen de los métodos o funciones que usualmente se suelen utilizar para trabajar con sockets UDP.

objetivo	método	Parámetros más significativos
Crear socket (vista en práctica anterior)	**socket ()**	-Tipo de socket a crear (TCP/UDP). -Devuelve un objeto de tipo socket con un atributo con el nuevo descriptor de fichero.
Asignar dirección local a un socket (vista en práctica anterior)	**bind ( )**	-Dirección Local (dirección IP y número de puerto) a la que quiero vincular el socket. (si la dirección ya está en uso obtendrás un error)
Enviar datos a un destino	**sendto()**	-Mensaje a enviar -Dirección del socket remoto al que dirigir los datos (IP y puerto).
Recibir datos de cualquier origen	**recvfrom()**	-Número de bytes a leer. (tamaño del buffer) Devuelve una pareja (string, address) con los bytes leídos y la dirección del origen (tupla IP y puerto).
Cerrar socket	**close()** **shutdown()**	-Cierra el socket y libera todos los recursos asociados. Toda operación posterior sobre el socket dará error. En shutdown se permite el cierre parcial (TX o RX) pero no se liberan recursos.
Otras	**getsockopt()** **setsockopt()**	Permiten leer y establecer algunos aspectos de configuración del socket (p.e. tamaño de los buffers de TX y RX, temporizadores, reutilización de puertos ya en uso, reacción ante errores, etc.. ver en el manual).

Antes de continuar, puedes arrancar un navegador y abrir la página https://docs.python.org/2/library/socket.html#socket.SOCK_DGRAM donde tienes una lista exhaustiva de todos los métodos de clase y de instancia que implementan los sockets en Python 2.7. Échale un vistazo durante 5 minutos antes de continuar con la práctica.

FLUJO DE LA COMUNICACIÓN CLIENTE/SERVIDOR UDP

La siguiente figura muestra la secuencia habitual en una comunicación entre un proceso cliente y otro servidor a través de sockets de tipo UDP.



<u>En el lado del cliente</u>, los principales pasos son:

1. **Creación de un socket (punto final de una comunicación)**. El cliente comienza solicitando al S.O. la creación de un socket udp (ver en la práctica anterior).

   (opcional) Si lo necesitase, el cliente podría asignar una dirección a su socket usando el método bind. Sin embargo, este no suele ser el caso (en clientes) y será el sistema operativo el que le asigne la dirección local de forma automática (ver en la práctica anterior).

2. **Envío del mensaje (A-PDU) al servidor**. Podemos enviar datos a cualquier otro proceso que use sockets UDP siempre que sepamos la dirección del socket remoto destinatario del mensaje.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
#direccion remota (destino del mensaje)
host = '127.0.0.1';
port = 8888;
#creo el mensaje
mensaje = "hola";
#envio los 4B del mensaje (A-PDU) al destino 127.0.0.1:8888
s.sendto(mensaje, (host,port));
```

(en el ejemplo anterior, si usas Python3 deberías escribir `mensaje.encode()` en el método sendto para evitar un error)

3. **Recepción de la respuesta del servidor**. Para recibir los mensajes que lleguen a nuestro socket local simplemente debemos hacer uso del método recvfrom indicándole el número máximo de bytes a recibir. La ejecución se bloquea hasta que llegue algún mensaje y nos devolverá como resultado tanto el mensaje recibido como la dirección del socket remoto (remitente).

```
msg, (addr, port) = s.recvfrom( 100 );
print msg, "recibido del remitente", addr,port;
```

4. **Cerrar el punto de comunicación abierto** (cerrar el socket). Antes de terminar el programa simplemente ejecuta **close**(); Si utilizas la función shutdown() recuerda que no libera los recursos asociados al socket por lo que en cualquier caso debes usar close().

En el lado del servidor, los principales pasos son:

1. **Creación de un socket**. (ya visto)
2. **Asociación del socket a una dirección**. El socket del servidor debe estar asociado a una dirección establecida por el diseñador de la aplicación distribuida y conocida por los clientes. Tal y como se vio en la práctica anterior, para ello se usa bind(). Por ejemplo `s.bind(('0.0.0.0', 23000))` asigna al socket `s` cualquier dirección IP de la máquina donde se ejecuta y el puerto 23.000.

3. **Recepción de mensaje**. (ya visto)
4. **Envío de mensaje**. (ya visto)
5. ir al paso 3.
6. Al terminar el programa - **cierre del socket**. (ya visto)

CÓDIGOS DE EJEMPLO DE UN CLIENTE Y DE UN SERVIDOR DE ECO (v1.0) : básico

A continuación abre en dos ventanas distintas los códigos de los ficheros cliente_eco_v1.py y servidor_eco_v1.py que puedes encontrar en el directorio de la práctica. Examina durante al menos 5 minutos ambos códigos.

- **Código del servidor**. El servidor crea un socket udp y le asigna la dirección *:44444 (i.e. cualquier dirección IP del equipo y el puerto 44.444). Después entra en un bucle en el que (a) lee un mensaje y (b) responde al origen (socket remoto) con el mismo mensaje.

```
import socket
import sys

HOST = ''   # Symbolic name meaning all interfaces (0.0.0.0)
PORT = 44444 # Arbitrary non-privileged port

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind socket to local host and port
s.bind((HOST, PORT))

#now the infinite loop. I reply to each message received.
while 1:
    # receive data from client (data, addr)
    d = s.recvfrom(1024)
    data = d[0]
    addr = d[1]
    # stopping when '.' is received
    if data.strip()=='.':
        break
    # otherwise just reply
    reply =  data

    s.sendto(reply , addr)
    print 'Message[' + addr[0] + ':' + str(addr[1]) + '] - ' + data.strip()
```

```
s.close()
```

Listado 1. Programa servidor (servidor_eco_v1)

Ejercicios:

1) El programa netcat (comando ncat o nc según el S.O., descárgalo de Internet si no lo tienes instalado) es un cliente de terminal remoto similar a telnet que puede usar tanto udp como tcp para enviar mensajes introducidos por el teclado a cualquier servidor. Abre dos terminales nuevos. En uno ejecuta el servidor anterior. En el otro ejecuta netcat como cliente udp: (suponiendo que el servidor se ejecuta en 127.0.0.1:44444)

   ```
   %> ncat 127.0.0.1 44444 –u -v.
   ```

   Escribe mensajes a través de netcat y comprueba cómo llegan al servidor y éste le responde. Abre un nuevo terminal y ejecuta otro nuevo cliente de netcat (dos clientes simultáneos contra el mismo servidor). Como vemos, un servidor udp sirve a varios clientes con el mismo socket. Comprueba cómo a cada cliente, el S.O. le ha asignado un número de puerto diferente. Para parar tanto netcat como el servidor tendrá que utilizar control+c.

2) En un terminal nuevo ejecuta otro servidor más (además del servidor que ya estás ejecutando). ¿Da error en la ejecución?. Eso es porque estás intentando asignar una dirección ya utilizada con bind (el mismo número de puerto en ambos servidores). ¿Cómo podrías hacer que dos servidores se ejecutasen a la vez?

- **Código del cliente**. El cliente lee el destino (dirección y puerto) desde los argumentos en línea de comandos (necesitas importar el módulo sys). Después entra en un bucle infinito en el que se pide al usuario que introduzca una frase por teclado. Al terminar la frase se la envía al servidor y se queda bloqueado a la espera de recibir la respuesta.

```
import socket
import sys

#check command line arguments (similar to C language)
if len(sys.argv) != 3:
    print 'Usage: python %s <HostName> <PortNumber>' % (sys.argv[0])
    sys.exit();

#get hostname and port number from command line. NOTE: port needs to be integer.
host=sys.argv[1]
port=int(sys.argv[2])  # port number should be an integer.
```

```
# create dgram udp socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

while(1) :
    msg = raw_input('Enter message to send : ')

    s.sendto(msg, (host, port))

    # receive a message (< 1024 bytes) from server (data, addr)
    d = s.recvfrom(1024)
    reply = d[0]
    addr = d[1]

    print 'Server reply : ' + reply

s.close()
```

- **Listado 2. Programa cliente (cliente_eco_v1)**

Ejercicios:

3) Ejecuta el cliente en un par de terminales y comprueba que funcionan. Si tienes algún compañero cerca o estás en el CdC puedes ejecutar clientes y servidor en equipos distintos.

4) Cambia el código del cliente para que se salga del bucle y termine cuando reciba '.' (p.ej. haga `break` dentro del bucle cuando el mensaje recibido sea '.'). Ello implica cambiar también el servidor para que envíe dicho carácter después de responder a un cliente.

5) Usa wireshark para capturar los mensajes intercambiados por cliente y servidor. (consejo, filtra por las direcciones IP y puertos afectados). Examina las cabeceras UDP y comprueba que el campo tamaño del segmento es correcto en función del mensaje de aplicación que encapsula.

CÓDIGOS DE EJEMPLO DE UN CLIENTE Y DE UN SERVIDOR DE ECO (v2.0) : excepciones

En el ejercicio 2, algo falló en tiempo de ejecución y obtuvimos una excepción. Puedes leer cómo Python hace uso de las excepciones en: https://docs.python.org/3/tutorial/errors.html . En nuestro caso, estaremos interesados en capturar las excepciones que se produzcan al utilizar los métodos de un socket (p.ej. al usar bind sobre una dirección ya ocupada o incorrecta, un socket que no se crea, un mensaje que no puede ser enviado, etc.) y así poner mostrar por pantalla un mensaje personalizado y controlar el flujo de ejecución del programa. El módulo socket exporta `socket.error` para manejar este tipo de excepciones. El valor que devuelve dice qué ha fallado o bien una pareja `(errno,string)` que representa el error devuelto por la llamada al sistema. Por ejemplo:

```
try:
    s.bind((HOST, PORT))
except socket.error , msg:
    print 'Fallo en bind. Codigo de Error : ' + str(msg[0]) + ' Mensaje ' + msg[1]
    sys.exit()
```

Examina el código de la versión 2 del cliente (v2) y del servidor (v2) durante 5 minutos.

Ejercicios:

6) Ejecuta la versión 2 del servidor y del cliente. Prueba a generar alguna excepción. Por ejemplo, haciendo bind a una dirección ya en uso al arrancar el servidor en dos terminales.

7) Los sockets, por defecto, se bloquean hasta que se completa la ejecución de algunos métodos (p.ej. en la recepción de mensajes con `recvfrom`). Este comportamiento puede ser cambiado para que un socket no se bloquee nunca. Para ello se usa el método de objeto `setblocking(0)` (ver en https://docs.python.org/2/library/socket.html#socket.SOCK_DGRAM). Reescribe el código del servidor (en un nuevo fichero llamado `servidor_eco_v25.py`) para que no se bloquee y capturar la excepción si no se recibe nada justo al ejecutar el método `recvfrom`, imprimiendo el mensaje "recepción fallida".

CÓDIGOS DE EJEMPLO DE UN CLIENTE Y DE UN SERVIDOR DE ECO (v3.0) : timeout

En el ejemplo del ejercicio anterior, la excepción es producida porque el método `recvfrom` no encuentra nada que leer en el momento de su ejecución. En la versión original del servidor (v2) el método no producía excepciones porque el socket se quedaba bloqueado *ad infinitum* hasta que se recibía algo. Este es el comportamiento por defecto en los sockets. Sin embargo, en nuestros programas hay veces que deseamos establecer un plazo máximo de recepción de mensajes, pasado el cual se genere alguna excepción. Por ejemplo, el cliente podría cerrarse si el usuario no escribe nada en 30 segundos, o si el servidor no le responde en menos de 5 segundos.

Para realizar esto dispones de dos alternativas:

- Select: genérica, utilizable con cualquier dispositivo de I/O (socket o fichero).
- Método settimeout() de un objeto socket. Establece un tiempo máximo de bloqueo para las operaciones del objeto socket.

(a) Uso de select

Python, al igual que C, ofrece la posibilidad de entrada/salida asíncrona entre nuestro programa y el exterior a través del módulo select. El módulo select permite **multiplexar eventos** de diversos tipos para uno o varios sockets (o descriptores de ficheros u objetos de tipo file). Por ejemplo, se le puede ordenar a select que nos notifique cuándo un socket tiene datos disponibles, o cuándo acepta la escritura de nuevos datos, o cuándo ocurre un error. Python utiliza listas para representar el conjunto de descriptores u objetos que debe

monitorizar y devuelve aquellos descriptores u objetos que satisfacen el evento correspondiente. Por ejemplo:

```
import sys
import select
import time

rlist, wlist, elist = select.select( [sys.stdin], [], [] )
print str(time.time())+ str(rlist)
print rlist[0].read()
```

Los argumentos pasados a select son 3 listas (dos de ellas vacías) que representan eventos de lectura, eventos de escritura, y eventos de error. En cada lista nosotros le pasamos tantos descriptores u objetos a monitorizar como queramos. En nuestro ejemplo sólo le hemos pedido que monitorice el objeto que representa a la entrada estándar [sys.stdin][1] en la lista de eventos de lectura. El método select devuelve tres listas que contienen aquellos objetos cuyos eventos asociados se han cumplido. En nuestro ejemplo, rlist debería contener [sys.stdin] cuando existan datos disponibles para leer en stdin (o sea, cuando se toque alguna tecla). Estos datos son leídos posteriormente con el método read() del objeto (ver https://docs.python.org/2.4/lib/bltin-file-objects.html).

Select también nos permite establecer un plazo de espera máximo para que ocurra un evento. Este plazo se escribe como cuarto argumento. Por ejemplo, podemos poner un plazo máximo de 3 segundos para que el usuario escriba algo simplemente con:

```
rlist, wlist, elist = select.select( [sys.stdin], [], [] ,3)
```

Prueba a cambiar el ejemplo anterior para exigir que el usuario escriba algo en menos de 3 segundos. Como puedes comprobar, transcurridos 3 segundos la función select devuelve una lista vacía y continúa la ejecución del programa.

Examina el siguiente ejemplo: (versión simplificada del servidor, versión 3)

```
import socket
import sys
import select

# Create two sockets in the server
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s2 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind sockets: s to address: *:8888 , s2 to address *:9999
s.bind(('', 8888))
s2.bind(('', 9999))

while 1:

   # Await a read event
    rlist, wlist, elist = select.select( [s, s2], [], [], 5 )
```

[1] Recuerda que cada proceso usa por defecto al menos 3 descriptores de ficheros (entrada estándar (0), salida estándar (1) y salida de errores (2)). Cada nuevo fichero abierto, o cada nuevo socket creado dará lugar a un nuevo descriptor de fichero (objeto).

```
      # Test for timeout
    if [rlist, wlist, elist] == [ [], [], [] ]:
        print "Han pasado 5 seg sin recibir nada por ningun socket.\n"
    else:
    # Loop through each socket in rlist, read and print the available data
        for sock in rlist:
            data, addr = sock.recvfrom( 1024 )
            print 'recibido en mi socket ' + str(sock.getsockname()) + '
desde el origen ' + addr[0] + ':' + str(addr[1]) + ': ' + data
            reply = data
            sock.sendto(reply , addr)

s.close()
s2.close()
```

- **Listado 3. Programa servidor simplificado (servidor_eco_v3)**

Como se puede apreciar, select vigila durante 5 seg si ocurre algún evento de llegada de datos en cualquiera de los dos sockets creados (i.e. multiplexión). En caso de que ocurra (p.ej. se reciben datos en s ó en s2) éstos se imprimen y se devuelve la respuesta al origen (servicio de eco). Si expiran los 5 segundos sin que ocurra ningún evento de lectura, el servidor imprime un mensaje en pantalla.

Ejercicios:

8) Usar dos terminales con netcat y probar el funcionamiento del servidor (cada terminal envía a cada puerto del servidor).

9) Cambiar el servidor (v3) para que cuando pasen 20 segundos seguidos sin recibir nada, cierre ambos sockets y salga del programa. (llámalo servidor_eco_v3_9.py)

10) Cambiar el servidor (v3) para que duerma 3 segundos antes de responder con el eco. Importa el modulo time y usa time.sleep(3) para dormir 3 segundos antes de responder (llámalo servidor_eco_v3_10.py). Comprueba su funcionamiento con el cliente(v2). Después, cambia el cliente usando select para que espere al mensaje de eco del servidor no más de X segundos (llámalo cliente_eco_v3_10). Si no llega respuesta del servidor, imprime un mensaje en pantalla y termina su ejecución. Prueba con diferentes valores de X y el servidor que acabas de modificar (v3_10). Por ejemplo, para X= 1, no debería recibir la respuesta y salirse. Para X=4 todo debería ir bien.

(b) Uso de settimeout

Esta es otra forma alternativa para establecer un temporizador de recepción de mensajes. Con este método simplemente le damos un plazo máximo a un socket para que cualquier método bloqueante sea ejecutado. Si no se ejecuta, salta una excepción de tipo *timeout*. (puedes leer más en https://docs.python.org/2/library/socket.html).

Un ejemplo parecido al código anterior (con sólo un socket) sería:

```
import socket
import sys

TIMEOUT = 5.0

try :
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
except socket.error, msg :
    print 'Failed to create socket. Error Code : ' + str(msg[0]) + ' Message ' + msg[1]
    sys.exit()

# Bind socket to address: *:8888
try:
    s.bind(('', 8888))

except socket.error , msg:
    print 'Bind failed. Error Code : ' + str(msg[0]) + ' Message ' + msg[1]
    sys.exit()

# set timeout TIMEOUT (see above) for socket s blocking operations
s.settimeout(TIMEOUT)

try:
    while 1:
        data, addr = s.recvfrom(1024)
        print 'recibido desde ' + addr[0] + ':' + str(addr[1]) + ': ' + data
        reply = data
        sock.sendto(reply , addr)

except socket.timeout,msg:
    print '** Exception : socket' + str(s.getsockname()) + msg + '  after ' + str(TIMEOUT) + ' seg.'
    s.close()
    sys.exit()
s.close()
```

- **Listado 4. Programa servidor simplificado (servidor_eco_v3b)**

En el código puedes comprobar cómo se puede establecer un temporizador de TIMEOUT segundos (constante de tipo float) asociado únicamente al socket s. A continuación puedes intentar realizar el siguiente ejercicio:

Ejercicio:

11) Intenta cambiar el código del servidor_eco_v3b para que sea similar al de v3. Es decir, que cree dos sockets y que se pueda recibir por cualquiera de ellos, mostrando un mensaje por pantalla si pasan más de 5 segundos sin recibir nada. Inténtalo al menos durante 10 min.

Tras intentar el ejercicio anterior, lo más probable es que haya llegado a la conclusión de que select permite la vigilancia simultánea de varios sockets (multiplexión de eventos) pero hacer esto mismo es imposible con settimeout ya que la orden de vigilancia sólo se hace a un socket que, a su vez, se bloquea hasta que expira el temporizador.

FINALIZANDO LA PRÁCTICA: EJERCICIOS FINALES

Te proponemos dos actividades o tareas de auto-evaluación de los conocimientos y destreza adquiridos.

Ejercicio Final 1: Crea un servidor de eco en un puerto aleatorio en el rango 20.000 – 20.100. (importa el módulo random y usa el método randint()). Después, debes hacer un cliente que, dada la dirección IP del servidor, haga un escaneo de puertos (i.e. prueba de forma secuencial todos los números de puertos posibles) e imprime el puerto en el que el servidor está activo. Mira la ayuda de random en (https://docs.python.org/2/library/random.html). Puedes hacer que el cliente espere la respuesta del servidor durante 1 segundo o menos (p.ej. con select o settimeout(0.5)).

Ejercicio Final 2: Hacer un servidor que cree un fichero y escriba en él una nueva línea con cada mensaje recibido de un cliente (por ejemplo el cliente v2 o usando netcat). El servidor terminará el fichero cuando lleve más de 5 segundos sin recibir nada. Puedes mirar en https://www.tutorialspoint.com/python/python_files_io.htm para leer más sobre el uso ficheros.

¿Qué deberías saber a partir de ahora?

Deberías haber aprendido a utilizar el servicio de transporte UDP en tus aplicaciones.
En particular:

- Crear un socket de tipo UDP y asociarlo a una dirección local cuando sea necesario.
- Enviar mensajes a cualquier destino a través de tu socket UDP
- Recibir mensajes de otros y responder a través de tu socket UDP
- La estructura básica de un cliente UDP
- La estructura básica de un servidor UDP
- Capturar las excepciones producidas por el uso de sockets
- Establecer temporizadores para la recepción de mensajes Hacer una aplicación distribuida completa (como la de los ejercicios finales).