PRÁCTICA 4: Cliente-servidor TCP

El objetivo de esta práctica es que te familiarices con el desarrollo de aplicaciones sencillas que usan los servicios de transporte del protocolo TCP. 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.

Recuerda que cuando la comunicación entre dos procesos de aplicación usa TCP, el servidor crea un socket y lo asocia a una dirección local (IP:puerto) conocida por el cliente. A continuación, pone dicho socket en modo escucha (estado *listening*). Cuando se recibe y se acepta una petición de conexión de un cliente, el servidor crea automáticamente otro socket nuevo (de conexión) a través del cual se comunica con el cliente que realizó la petición de conexión. El el lado del cliente, simplemente se crea un socket y se pide conexión con el servidor deseado. Quizás quieras repasar las prácticas anteriores antes de continuar para recordar cómo se direccionan los procesos de aplicación y sus sockets asociados.

INTRODUCCIÓN AL USO DE SOCKETS TCP

En la práctica anterior aprendiste cómo programar aplicaciones distribuidas UDP. La mayoría de métodos practicados son también válidos para su uso con TCP.

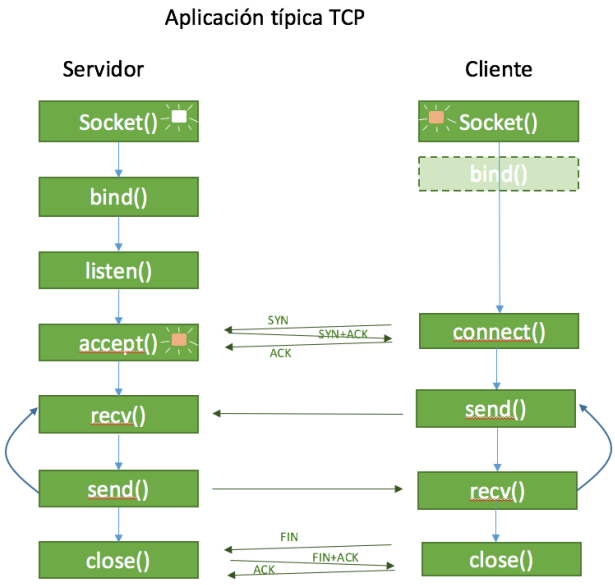
objetivo	método	Parámetros más significativos
Conectar con un servidor	**connect()**	- dirección del servidor remoto (dirección IP y número de puerto)
Escuchar peticiones de conexión	**listen()**	- número de peticiones de conexión que puede almacenar.
Aceptar una petición de conexión recibida	**accept()**	- no recibe parámetros (en python será un método del socket de escucha). **Devuelve un nuevo socket** (i.e. crea un nuevo socket, de conexión)
Enviar datos al socket	**send()**	- Mensaje a enviar (string). El socket debe estar previamente conectado con otro socket remoto. Devuelve el número de bytes enviados que debería ser comprobado ya que puede que no envíe todo el mensaje y sea necesario volver a enviar más bytes. Es responsabilidad de la aplicación comprobar que la llamada ha enviado todos los bytes deseados.
Enviar datos al socket	**sendall()**	- Mensaje a enviar (string). Similar a send() pero el método se asegura de que todos los datos del mensaje son enviados.
Recibir datos del socket	**recv()**	-Número máximo de bytes a leer. (tamaño del buffer) Devuelve un string que representa los bytes leídos.

Antes de continuar, puedes arrancar un navegador y abrir la página https://docs.python.org/2/library/socket.html donde tienes una lista exhaustiva de todos los métodos de clase y de instancia que implementan los sockets.

FLUJO DE LA COMUNICACIÓN CLIENTE/SERVIDOR TCP

La siguiente figura muestra la secuencia habitual en el uso de los métodos de la tabla anterior en una comunicación entre un proceso cliente y otro servidor que utilizan los servicios de transporte del protocolo TCP.

En el lado del cliente, los principales pasos son:



1. **Creación de un socket.** El cliente comienza solicitando al S.O. la creación de un socket de tipo flujo (SOCK_STREAM).

   (opcional) bind (no es necesario. Ver en la práctica anterior).

2. **Conexión con el socket del servidor**. El cliente solicita el establecimiento de una conexión con otro socket remoto. La ejecución de este método se bloquea hasta que el servidor acepte o rechace esta conexión.

3. **Envío de datos al servidor**. A través del socket, ya conectado, podemos enviar y recibir datos. Podemos hacerlo con la función `send()` o `sendall()`. Normalmente preferiremos hacerlo con `sendall()` ya que nos asegura que todos los bytes entregados en la llamada son enviados. Con el método `send()` esto no estaría asegurado y, por ello deberíamos verificar que todos los bytes hayan sido efectivamente enviados y completar con nuevos envíos en caso de que todavía falten datos por enviar.

4. **Recepción de la respuesta del servidor**. Para recibir los mensajes que lleguen a un socket ya conectado, simplemente debemos hacer uso del método `recv` indicándole el número máximo de bytes a recibir. La ejecución se bloquea hasta que haya bytes en el buffer de recepción del socket y nos devolverá como resultado los datos recibidos por el socket hasta un máximo indicado como parámetro.

```
data = socketconexion.recv( 100 );
print data, "recibidos del socket de conexion socketconexion";
```

5. **Cerrar el punto de comunicación abierto** (cerrar el socket). (ver práctica anterior).

A continuación puedes ver un ejemplo de una comunicación TCP desde un cliente. En particular, es un ejemplo sacado de http://www.binarytides.com/python-socket-programming-tutorial/ en el que el cliente conecta con un servidor web, le envía un mensaje de petición del protocolo HTTP y recibe la respuesta del servidor.

```
#Socket client example in python

import socket   #for sockets
import sys  #for exit

#create an INET (i.e. IPv4), STREAMing (i.e. TCP) socket
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error:
    print 'Failed to create socket'
    sys.exit()

print 'Socket Created'

host = 'www.google.com';
port = 80;


#Connect to remote server
s.connect((host , port))

print 'Socket Connected to ' + host

#Send some data to a remote web server
message = "GET / HTTP/1.1\r\n\r\n"

try :
    #Set the whole string
    s.sendall(message)
except socket.error:
    #Send failed
    print 'Send failed'
    sys.exit()

print 'Message send successfully'

#Now receive data
reply = s.recv(4096)

print reply
```

Prueba a realizar conexiones con diferentes servidores web enviándoles distintos mensajes de petición (p.ej. el del código anterior, o "HEAD / HTTP/1.0\r\n"). Puedes hacerlo modificando el código, o bien directamente con el programa netcat usado en la práctica anterior (recuerda no usar la opción –u ya que ahora queremos una conexión tpc).

Ejercicio:

1) Partiendo del código del cliente udp (v2) de la práctica anterior, desarrolla un cliente de eco que utilice tcp (cliente_eco_tcp_v1). Para comprobar el correcto funcionamiento de tu cliente, de momento, puedes usar el programa netcat escuchando en modo servidor (opción –l). Comprueba cómo netcat recibe el mensaje de tu cliente y comprueba cómo tu cliente recibe lo que escribas directamente en netcat.

En el lado del servidor, los principales pasos son:

1. **Creación de un socket TCP**. (ya visto.)

2. **Asociación del socket a una dirección**. (ya visto).

3. **Poner el socket en estado de escucha** de peticiones de conexión. Para ello se utiliza el método listen(). Este método recibe como único parámetro un entero que indica el número máximo de peticiones de conexión pendientes de servir que podrían aceptase temporalmente.

4. **Aceptación** de una nueva conexión. Para ello se usa el método `accept()`. Es un método bloqueante hasta que recibe alguna petición de conexión. Al recibirla, **el método crea y devuelve un nuevo socket** (podemos verlo como un *socket conectado* utilizable únicamente para hablar con ese cliente) y la dirección remota que hizo la solicitud de conexión.

5. **Recepción de mensaje** a través del socket conectado. (ya visto)
6. **Envío de mensaje**. a través del socket conectado. (ya visto)
7. Al terminar el programa - **cierre de sockets**. (ya visto)

Examina ahora el código de éste servidor de eco, identificando los pasos anteriores.

```
import socket
import sys

HOST = ''   # Symbolic name meaning all available interfaces
PORT = 5000 # Arbitrary non-privileged port for server

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

s.setsockopt( socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

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

s.listen(10)
print 'Socket now listening'

#wait for connection and return a new socket and the remote address
conn, addr = s.accept()
print 'Connected with ' + addr[0] + ':' + str(addr[1])

#now talking to the client via conn socket
while 1:

    data = conn.recv(1024)
    reply = data
    if data=='.':
        break

    conn.sendall(reply)
```

```
conn.close()
print 'Disconnected ' + addr[0] + ':' + str(addr[1])
s.close()
```

Listado 1. Programa servidor (servidor_eco_tcp_v1)

Como puede apreciar en el código, tras la creación de un socket tcp se establece la opción SO_REUSEADDR en el mismo. La opción SO_REUSEADDR permite decirle al sistema operativo que puede reutilizar de forma inmediata un socket que todavía no ha sido completamente liberado.

Después, el socket creado se pone en modo escucha con capacidad para recibir nuevas peticiones de conexión (hasta un máximo de 10), que serán almacenadas temporalmente hasta que puedan ser atentidas.

Posteriormente el código se queda bloqueado en el método `accept()` esperando la recepción de una petición de conexión de algún cliente. Al recibirse, el método se desbloquea y **devuelve**:

- **Un nuevo objeto socket**. Este es un socket ya conectado con el cliente (socket de conexión o conectado) y será utilizado para la comunicación con dicho cliente.
- **La dirección del extremo remoto** que realizó la solicitud de conexión. (i.e. dirección IP y puerto).

A partir de este momento, al usar el nuevo socket de conexión se intercambiarán datos con dicho cliente. Hasta que el cliente no envíe *nada* (o se pulse *control+c*) no se cerrará la conexión.

Ejercicio:

2) Ejecuta el código del servidor (servidor_eco_tcp_v1). Ejecuta en otro terminal el código de tu cliente del ejercicio anterior (cliente_eco_tcp_v1) y comprueba que funciona. Mientras estás enviando mensajes al servidor abre otro terminar y ejecuta simultáneamente otro cliente. ¿Cómo podrías hacer que funcione?.

   Examina el código del servidor (servidor_eco_tcp_v1_2). Prueba el funcionamiento correcto con un par de clientes (i.e. cuando un cliente envía *control+c* le toca el turno al otro). Como ves, el servidor acepta conexiones pendientes reutilizando su único socket de conexión por turnos.

3) Crea otra nueva versión del servidor (servidor_eco_tcp_v1_3) que cierre la conexión con el cliente después de envíar la primera respuesta, o bien si no se recibe nada de un cliente en menos de 5 segundos (usando el método `settimeout` o `select` vistos en la práctica anterior). Tras cerrar la conexión con un cliente, el servidor debe aceptar nuevas peticiones de conexión. Modifica también el cliente para que se cierre después de recibir la primera respuesta. Prueba el funcionamiento, ¿es más fácil servir a varios clientes?

ATENDIENDO A VARIOS CLIENTES EN PARALELO (v2)

En ocasiones desearemos hacer un servidor que atienda a varios clientes de manera más o menos simultánea. Para esto existen varias opciones:

(a) **Usar conexiones no persistentes**. Esto es, realizar la secuencia aceptar + servir+ cerrar con cada petición de un cliente (esto es lo que hemos realizado en el ejercicio anterior). Como el socket de escucha nos permite almacenar las peticiones de conexión de otros clientes, en cuanto se cierra el socket de conexión de un cliente se puede atender inmediatamente al siguiente cliente. De esta forma, si el tiempo de servicio es pequeño, puede dar la impresión de servir en "paralelo". Sin embargo, si el diálogo con el cliente incluyese varias rondas de peticiones y respuestas, el cliente necesitaría realizar un nueva conexión para cada petición.

(b) **Utilizar el método select** para multiplexar varios sockets. Se puede vigilar la llegada de un evento de lectura a un conjunto de sockets. Cuando llega el evento se distingue si llega al socket de escucha (en cuyo caso se crea un socket de conexión y se añade a la lista a vigilar), o si llega a un socket de conexión (en cuyo caso se sirve al cliente). De esta forma se pueden mantener simultáneamente varios sockets de conexión en paralelo. Quizás quieras repasar el uso de select en la práctica anterior.

(c) **Utilizar programación multi-hilos o programación multi-proceso**. Los hilos de ejecución (thread) son una forma de dividir un programa en secuencias de instrucciones que pueden ser ejecutadas como tareas de forma independiente por el sistema operativo. La idea aquí es que cada vez que aceptamos una nueva conexión podríamos crear un nuevo hilo (o proceso) para servirla. En este curso no vamos a exigir la programación con esta opción, pero si tienes interés puedes ver un ejemplo sencillo de nuestro servidor de eco en http://www.binarytides.com/python-socket-programming-tutorial/ (sección handling connections).

(b) Uso de Select() para atender a varios clientes.

El código que sigue a continuación es un extracto del fichero servidor_tcp_v2.py donde, para mayor claridad, no se muestran las líneas de creación y direccionamiento del socket.

```
s.listen(10)
print 'Socket now listening'

descriptors = [s] #list of sockets created

#infinite loop
while 1:
    #list of sockets to watch for read events
    (sread, swrite, sexc) = select.select( descriptors, [], [] )

    for sock in sread:
        if sock == s:   #s is the listening socket: this is a connect request
             conn, addr = s.accept()
             print 'Connected to ' + addr[0] + ':' + str(addr[1])
             descriptors.append(conn)   #sock added to the list to watch
        else:
            data = sock.recv(100)
            if data =='':
                host,port = sock.getpeername()
                print 'client left ' + str(host) + ':' + str(port)
                sock.close()
                descriptors.remove(sock)
            else:
```

```
                host,port = sock.getpeername()
                print 'client ' + str(host) + ':' + str(port) + 'data: ' +
str(data)
                reply = data
                sock.sendall(data)

s.close()
```

Listado 2. Extracto del programa servidor (servidor_eco_tcp_v2)

Tal y como se puede apreciar en las líneas mostradas, utilizamos una lista de sockets como soporte para la multiplexión a través de `select`. La lista comienza con sólo el socket de escucha. Si ocurre un evento de lectura en la lista, y éste ocurre en el socket de escucha, sólo puede tratarse de una petición de conexión. En tal caso aceptamos la conexión. Esta operación implica la creación de un nuevo socket conectado al cliente que envió la petición (en el código, el objeto `conn`). Por ello, lo añadimos a la lista de sockets utilizados. Si se recibe un evento de lectura y éste no ocurre en el socket de escucha sólo puede ocurrir en algún socket de conexión. Por ello realizamos el servicio de eco leyendo dicho mensaje y respondiendo al cliente. En el caso de que el mensaje recibido sea '' significa que el cliente quiere terminar (p.ej. el cliente ha escrito control+c) entonces simplemente se cierra dicho socket de conexión y se elimina de la lista.

Ejercicios:

4) Abre dos terminales clientes con la versión 1 (cliente_eco_tcp_v1.py). Ahora conéctalos con el servidor (servidor_eco_tcp_v2.py). ¿Funcionan en paralelo?. En un nuevo terminal usa `netstat` (ver práctica 2) y mira si identificas todos los sockets creados: tres en el servidor (1 de escucha y 2 de conexión), y uno (de conexión) por cada cliente. Dibuja un diagrama como los de las práctica 2 incluyendo los números de puerto.

5) Repite el ejercicio anterior. Abre Wireshark y captura el tráfico de cada conexión. Dibuja un pequeño esquema con los segmentos intercambiados en las fases de establecimiento, transferencia (envíe un pequeño mensaje) y liberación. Recuerda en Wireshark marcar la opción para que se vean los números de secuencia originales de las cabeceras de los segmentos en lugar de los números de secuencia relativos (ver https://wiki.wireshark.org/TCP_Relative_Sequence_Numbers). Comprueba que los números de secuencia y asentimiento coinciden con lo visto en la teoría en las tres fases de la conexión.

6) Intente crear una nueva versión del servidor donde se expulse a los clientes que no envían nada durante 5 seg ¿ha encontrado algún problema?.

TAMAÑO O LÍMITE DE LOS MENSAJES (v3).

TCP no tiene concepto de mensaje sino de flujo de bytes. Por ello, si un cliente enviase de forma seguida dos mensajes pequeños, el servidor podría recibir ambos con una sola

operación de lectura. Igualmente, si un cliente enviase más bytes de los que el servidor indica en la instrucción `recv` entonces los datos recibidos estarían incompletos.

Por ejemplo, examina los códigos:

cliente

```
import socket
import sys

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

s.connect(('127.0.0.1',5000))

print 'socket conectado'

msg1 = 'hola'
print 'envio: hola'
s.sendall(msg1)
msg2 = 'adios'
print 'envio: adios'
s.sendall(msg2)

s.close()
```

servidor

```
import socket
import sys
import time

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

s.bind(('',5000))

s.listen(10)
conn, addr = s.accept()

time.sleep(2) #duermo 2 segs

data = conn.recv(100)
print  data

s.close()
```

Para indicar al servidor cuántos bytes tiene un mensaje es necesario que nosotros encapsulemos el propio mensaje y añadamos una cabecera indicando el tamaño del mensaje. Por lo tanto es necesario que la A-PDU del cliente incluya un campo adicional a los datos que indique la propia longitud de los datos (p.ej. en bytes), y el servidor lea este campo para saber el tamaño exacto de los datos que tiene que leer.

La implementación de este protocolo de aplicación depende de cada uno. Nosotros os ofrecemos una implementación sencilla donde se incluye una cabecera de tamaño fijo en cada mensaje que indica el tamño del resto del mensaje (hasta 999999 bytes). Los códigos de la función de encapsulado y desencapsulado de los mensaje podrían ser :

Función de encapsular

```
def encapsula(mensaje):
    tam = len(mensaje)
    apdu = '{:0>6n}'.format(tam) +
':' + mensaje
    return apdu
```

Función desencapsular

```
def readmsg(conn):
    data = conn.recv(8)
    c = data.split(':')
    tamano = int(c[0])
    msg = c[1]
    pending = tamano - len(c[1])

    while(pending > 0):
        data = conn.recv(pending)
        msg = msg + data
        pending = pending - len(data)
    return msg
```

El código de encapsulación anterior simplemente añade un campo al mensaje que consiste en 6 caracteres que representan el tamaño del mensaje encapsulado en bytes. El separador entre este campo y los datos es el carácter ':'. El código de desencapsulación simplemente lee 8 bytes (de los que los 6 primeros son el tamaño, el séptimo es el separador y el último es el primer byte del mensaje encapsulado). Posteriormente, si el mensaje encapsulado tiene más de un byte, lee los restantes hasta que los obtiene todos. Uno podría inventarse

otras formas de realizar estas dos funciones dentro de su protocolo de nivel de aplicación (p.ej. sólo leer el tamaño la primera vez).

Ejercicio:

7) Examine atentamente el código del cliente (cliente_tcp_v3) y del servidor (servidor_tcp_v3). Haga pruebas con el cliente y el servidor. Una vez que verifique que funciona, cambie los códigos del cliente y el servidor para que funcionen como en la aplicación del servicio de eco. Realice en su código la captura de excepciones. Llame a los nuevos programas cliente_eco_tcp_v3_7 y servidor_eco_tcp_v3_7

FINALIZANDO LA PRÁCTICA: EJERCICIOS FINALES

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

Ejercicio Final 1: **"Whatsapp de grupo"**. Estudiar el código del servidor de chat del documento tutorial adjunto. Descargar el código y ponerlo en marcha. El ejercicio consiste en (a) comprender bien el funcionamiento del servidor; (b) Cambiar el servidor para que solicite un password a cada cliente (puede ser una palabra secreta escrita en el propio código) y para que expulse a los clientes que lleven más de 1 minuto inactivos (si fuera posible).

Ejercicio Final 2: **"copia remota de ficheros"**. Realizar un cliente que abra un fichero para lectura. Lea el fichero línea a línea y las envíe una a una al servidor TCP. El servidor debe ir escribiendo en un fichero el contenido recibido El resultado final debería ser que el fichero ha sido copiado desde el origen hasta el destino remoto donde se ejecuta el servidor. Haga una segunda versión donde el cliente lee todo el fichero y lo mete en un solo mensaje que encapsula con una cabecera similar a la de la última parte de la práctica. Después el cliente envía con un solo `sendall()` todo el contenido del fichero al servidor. El servidor debe recibir todo el fichero y grabarlo como un fichero local.

Ejercicio Final 3: Estudio del protocolo TCP. Capture con wireshark una transferencia de un fichero sobre TCP (p.ej la del ejercicio anterior, o bien la descarga del fichero alice.txt en un navegador). El fichero alice.txt se encuentra disponible en: http://www.umich.edu/~umfandsf/other/ebooks/alice30.txt . Después examine detenidamente en su captura toda la conexión TCP (establecimiento, transferencia, cierre). Examine los conceptos vistos en clase (transferencia fiable, control flujo, control de congestión). Por ejemplo, ¿ha obtenido algunas retransmisiones?,¿se ha llenado el buffer de recepción?, ¿se sigue la dinámica del arranque lento?, etc… Examine las gráficas que genera wireshark en su menú : estadísticas/tcpstream graphs . Allí puede examinar la secuencia temporal (Stevens) de números de secuencia y ACKs recibidos durante la transferencia, la variación del throughput o el tiempo de ida y vuelta a lo largo de la conexión.

¿Qué deberías saber a partir de ahora?

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

- Crear un socket de tipo TCP y asociarlo a una dirección local cuando sea necesario.
- Enviar mensajes a cualquier destino a través de tu socket TCP
- Recibir mensajes de otros y responder a través de tu socket TCP
- La estructura básica de un cliente y servidor TCP
- Servir a varios clientes en paralelo usando el método select.
- Establecer temporizadores para la recepción de mensajes
- Delimitar con un protocolo de aplicación el tamaño de los mensajes.
- Hacer una aplicación distribuida completa (como la de los ejercicios finales).
- Examinar la dinámica de tcp durante la fase de transferencia de datos.