Practica 1 python

ECHO CLIENT/SERVER

Alvaro Gala Martinez  
Adrián Pérez Fernández  
Andrés Casado García  
Daniel Montero García

# INTRODUCCION

Esta practica se basa en la implementación de dos partes (servidor y cliente) que se comunican entre si de tal manera que el servidor espera solicitudes del cliente, y cuando le llegan devuelve una respuesta. Para ello, el cliente debe conocer los parámetros (dirección IP y numero de puerto) para establecer la conexión con el servidor.

Para conseguir esta comunicación entre procesos vamos a usar sockets (que cuenta con su propia API), los cuales nos facilitan una forma de IPC para llegar a nuestro objetivo.

# UN POCO DE HISTORIA ACERCA DE LOS SOCKETS

Los sockets se usaron por primera vez a partir del año 1980 aproximadamente, en la universidad californiana de Berkeley. Allí implementaron los sockets para poder escribir programas capaces de comunicarse entre sí. Esta implementación tuvo lugar en el sistema operativo BSD (Berkeley Software Distribution), una variante de UNIX. Por ello, la primera API oficial lanzada en 1983 se llamó Berkeley sockets, y es la que vamos a usar para el desarrollo de esta práctica.

# SOCKET API

El modulo de socket para Python nos proporciona una interfaz a la API de los Berkeley sockets (a partir de ahora los denominaremos simplemente sockets)

Esta API ofrece las siguientes funciones y métodos:

socket(), bind(), listen(), accept(), connect(), connect\_ex(), send(), recv(), close()

Mas tarde veremos cómo funcionan en su aplicación en el código, ya que vamos a usar todas para implementar nuestros sockets.

# CODIGO COMUN

Vamos a analizar el código, empezando por la creación del socket la cual es común en ambas partes cliente y servidor. Para ello usamos la función socket() donde vamos a introducir dos parámetros necesarios, AF\_INET y SOCK\_STREAM:

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

En primer lugar, usamos el parametro socket.AF\_INET para especificar el dominio, es decir, la familia de protocolos que se utilizara para la comunicación. En este caso, AF\_INET representa IPv4.

En segundo lugar, usamos el parametro socket.SOCK\_STREAM para especificar el protocolo de transporte que se va a usar en la comunicación. SOCK\_STREAM representa el protocolo TCP, el cual usamos porque garantiza que los datos son entregados en el destino sin errores y en el mismo orden en el que salieron.

Una vez que el socket esta correctamente inicializado, hay que establecer la conexión, para ello indicamos un host (dirección IP) y un puerto:

server\_address = ('127.0.0.1', 52186)

A partir de ahora el código entre la parte del cliente y la del servidor cambia, por lo que vamos a analizarlo por separado, empezando por el servidor.

# SERVIDOR

El servidor es la parte que esta escuchando y esperando a recibir mensajes desde los clientes. Para ello usamos el siguiente código:

sock.bind(server\_address)

sock.listen()

La función bind() asocia el servidor la dirección indicada en el parámetro de entrada (la variable server\_adress que hemos declarado anteriormente), y la función listen() hace que el servidor se quede escuchando de cara a aceptar conexiones provenientes de los clientes.

Esta aceptación por parte del servidor se realiza con la función accept():

connection, client\_address = sock.accept()

Esta función espera a una conexión entrante. Cuando un cliente se conecta, el servidor devuelve un nuevo socket que representa la conexión y una tupla con la dirección del cliente. Como estamos usando IPv4, esta tupla esta formada por el host y el puerto.

Es importante tener en cuenta el hecho de que ahora tenemos un nuevo socket creado por la función accept(), que es el que va a usar el servidor para comunicarse con el cliente. Este socket es distinto del inicial (sock), el cual hacia de “oyente” para aceptar conexiones.

A continuación, entramos en un bucle infinito en el que el servidor se prepara para recibir datos del cliente con la función recv(). A esta función le tenemos que pasar un argumento que va a especificar la cantidad máxima de datos a recibir:

data = connection.recv(16)

Cuando recibe el mensaje, lo imprime por pantalla y se lo devuelve al cliente con la función sendall():

print('RECEIVED:', data.decode())

connection.sendall(data)

Al estar en un bucle infinito, el servidor permanece escuchando hasta que decidimos cancelar el proceso, entonces el servidor cierra la conexión con la función close():

connection.close()

# CLIENTE

Empezando a partir del código común con el servidor ya explicado, vemos como el cliente usa la función connect() para conectarse al servidor. El argumento que le pasamos es la dirección a la que hemos asociado el servidor anteriormente:

sock.connect(server\_address)

Una vez establecida la conexión entre servidor y cliente, comenzamos el envio de la información. Para ello, declaramos un string genérico y se lo enviamos al servidor con la función sendall():

message = 'This is the message. It will be repeated.'.encode()

sock.sendall(message)

Después, el cliente recibe la respuesta del servidor y continua la comunicación fragmentando el mensaje con el argumento pasado a la función recv():

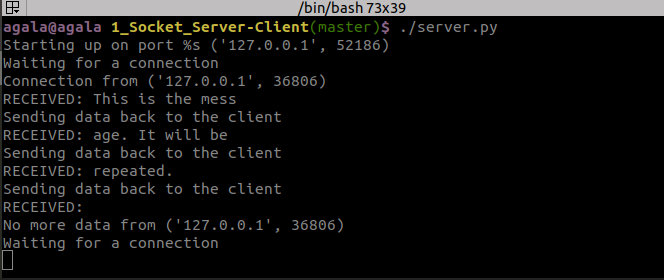
data = sock.recv(16)

Cuando se cumple la condición del bucle en el que estamos, es decir, el mensaje ya ha sido enviado por completo y no recibimos nada del servidor, terminamos cerrar el socket con la función close():

sock.close()

# SALIDAS

Servidor



Cliente

