Programación de socket en Python (Guía)

Table of Contents

Los sockets y la API de socket se utilizan para enviar mensajes a través de una red. Proporcionan una forma de comunicación entre procesos (IPC) . La red puede ser una red local lógica para la computadora, o una que esté conectada físicamente a una red externa, con sus propias conexiones a otras redes. El ejemplo obvio es Internet, a la que se conecta a través de su ISP.

Este tutorial tiene tres iteraciones diferentes para construir un servidor y un cliente de socket con Python:

1. Comenzaremos el tutorial mirando un servidor y un cliente de socket simples.
2. Una vez que haya visto la API y cómo funcionan las cosas en este ejemplo inicial, veremos una versión mejorada que maneja múltiples conexiones simultáneamente.
3. Finalmente, avanzaremos hacia la construcción de un servidor y un cliente de ejemplo que funciona como una aplicación de socket completa, con su propio encabezado y contenido personalizados.

Al final de este tutorial, comprenderá cómo usar las funciones y métodos principales en el módulo socket de Python para escribir sus propias aplicaciones cliente-servidor. Esto incluye mostrarle cómo usar una clase personalizada para enviar mensajes y datos entre puntos finales que puede desarrollar y utilizar para sus propias aplicaciones.

Las redes y los sockets son temas importantes. Se han escrito volúmenes literales sobre ellos. Si es nuevo en los sockets o en las redes, es completamente normal que se sienta abrumado con todos los términos y piezas. ¡Sé que lo hice!

Sin embargo, no se desanime. He escrito este tutorial para ti. Como hacemos con Python, podemos aprender poco a poco.

¡Empecemos!

Background

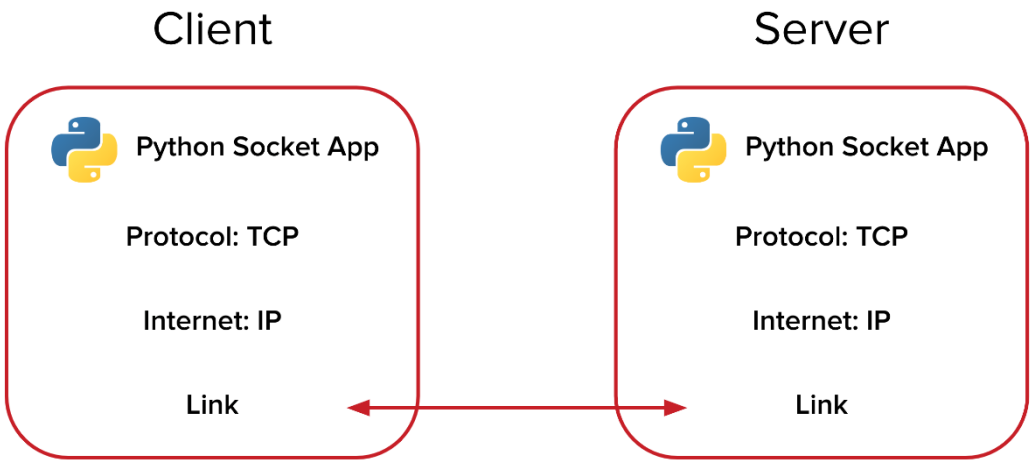
Los sockets tienen una larga historia. Su uso se originó con ARPANET en 1971 y luego se convirtió en una API en el sistema operativo Berkeley Software Distribution (BSD) lanzado en 1983 llamado Berkeley sockets .

Cuando Internet despegó en la década de 1990 con la World Wide Web, también lo hizo la programación de redes. Los navegadores y servidores web no eran las únicas aplicaciones que aprovechaban las redes recién conectadas y usaban sockets. Se generalizaron las aplicaciones cliente-servidor de todos los tipos y tamaños.

Hoy en día, aunque los protocolos subyacentes utilizados por la API de socket han evolucionado a lo largo de los años y hemos visto otros nuevos, la API de bajo nivel sigue siendo la misma.

El tipo más común de aplicaciones de socket son las aplicaciones cliente-servidor, donde un lado actúa como servidor y espera las conexiones de los clientes. Este es el tipo de aplicación que cubriré en este tutorial. Más específicamente, veremos la API de sockets para sockets de Internet , a veces llamados sockets Berkeley o BSD. También hay sockets de dominio Unix , que solo se pueden usar para comunicarse entre procesos en el mismo host.

Descripción general de la API de socket



El módulo de socket de Python proporciona una interfaz a la API de sockets de Berkeley . Este es el módulo que usaremos y discutiremos en este tutorial.

Las funciones y métodos de la API de socket principal en este módulo son:

- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `connect()`
- `connect_ex()`
- `send()`
- `recv()`
- `close()`

Python proporciona una API conveniente y consistente que se asigna directamente a estas llamadas al sistema, sus contrapartes C. Veremos cómo se usan juntos en la siguiente sección.

Como parte de su biblioteca estándar, Python también tiene clases que facilitan el uso de estas funciones de socket de bajo nivel. Aunque no se trata en este tutorial, consulte el módulo socketserver , un marco para servidores de red. También hay muchos módulos disponibles que implementan protocolos de Internet de nivel superior como HTTP y SMTP. Para obtener una descripción general, consulte Compatibilidad y protocolos de Internet .

TCP Sockets

Como verá en breve, crearemos un objeto de socket usando **`socket.socket()`**y especificaremos el tipo de socket como **`socket.SOCK_STREAM`**. Cuando lo hace, el protocolo predeterminado que se utiliza es el Protocolo de control de transmisión (TCP) . Este es un buen valor predeterminado y probablemente sea lo que desea.

¿Por qué debería utilizar TCP? El Protocolo de control de transmisión (TCP):

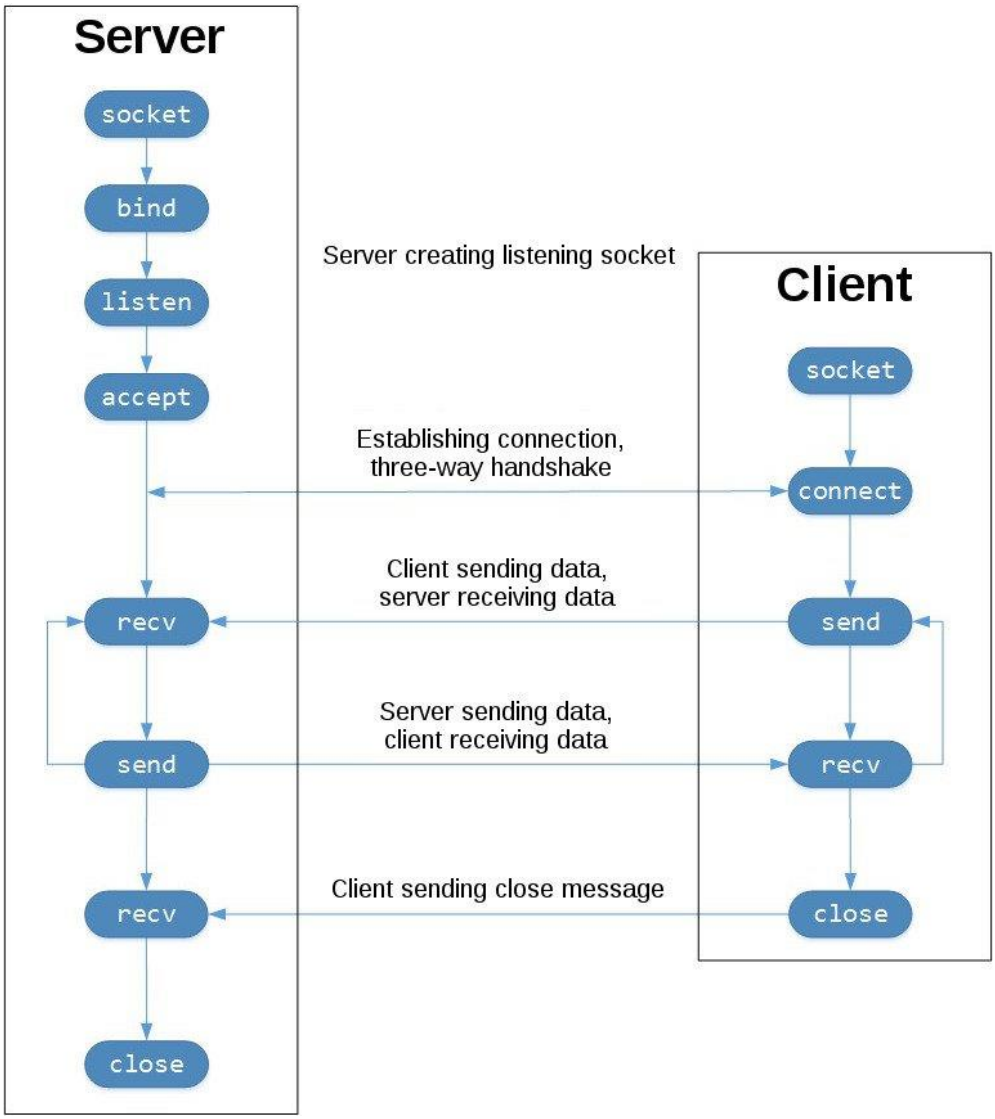
- **Es confiable: los** paquetes caídos en la red son detectados y retransmitidos por el remitente.
- **Tiene entrega de datos en orden:** su aplicación lee los datos en el orden en que fueron escritos por el remitente.

Por el contrario, los sockets del Protocolo de datagramas de usuario (UDP) creados con **`socket.SOCK_DGRAM`** no son confiables, y los datos leídos por el receptor pueden estar desordenados con respecto a las escrituras del remitente.

¿Porque es esto importante? Las redes son un sistema de entrega de mejor esfuerzo. No hay garantía de que sus datos lleguen a su destino o de que recibirá lo que se le envió.

Los dispositivos de red (por ejemplo, enrutadores y conmutadores) tienen un ancho de banda finito disponible y sus propias limitaciones inherentes al sistema. Tienen CPU, memoria, buses y búfer de paquetes de interfaz, al igual que nuestros clientes y servidores. TCP le evita tener que preocuparse por la pérdida de paquetes , los datos que llegan fuera de orden y muchas otras cosas que suceden invariablemente cuando se comunica a través de una red.

En el siguiente diagrama, veamos la secuencia de llamadas a la API de socket y el flujo de datos para TCP:



La columna de la izquierda representa al servidor. En el lado derecho está el cliente.

Comenzando en la columna superior izquierda, tenga en cuenta las llamadas a la API que hace el servidor para configurar un socket de "escucha":

- `socket()`
- `bind()`
- `listen()`
- `accept()`

Un socket de escucha hace exactamente lo que parece. Escucha las conexiones de los clientes. Cuando un cliente se conecta, el servidor llama **`accept()`**para aceptar o completar la conexión.

El cliente llama **`connect()`**para establecer una conexión con el servidor e iniciar el protocolo de enlace de tres vías. El paso del apretón de manos es importante ya que asegura que cada lado de la conexión sea accesible en la red, en otras palabras, que el cliente pueda llegar al servidor y viceversa. Puede ser que solo un host, cliente o servidor, pueda llegar al otro.

En el medio está la sección de ida y vuelta, donde se intercambian datos entre el cliente y el servidor mediante llamadas a **`send()`**y **`recv()`**.

En la parte inferior, el cliente y el servidor llaman a **`close()`** en sus respectivos sockets.

Echo Client and Server

Ahora que ha visto una descripción general de la API de socket y cómo se comunican el cliente y el servidor, creemos nuestro primer cliente y servidor. Comenzaremos con una implementación simple. El servidor simplemente enviará al cliente todo lo que reciba.

Servidor de eco

Aquí está el servidor `echo-server.py`:

```
#!/usr/bin/env python3

import socket

HOST = '127.0.0.1'  # Standard loopback interface address (localhost)
PORT = 65432        # Port to listen on (non-privileged ports are >
1023)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
```

```
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Nota: No se preocupe por comprender todo lo anterior en este momento. Están sucediendo muchas cosas en estas pocas líneas de código. Este es solo un punto de partida para que pueda ver un servidor básico en acción.

Repasemos cada llamada a la API y veamos qué está sucediendo.

`socket.socket()`crea un objeto de socket que admite el tipo de administrador de contexto , por lo que puede usarlo en una `with` declaración. No es necesario llamar al **`socket.close()`**(de igual manera que pasaba como con los ficheros que vimos el año pasado, al usar el with no hace falta cerrar el stream ya que se encarga el with una vez salimos de su bloque):

Los argumentos pasados para `socket()` especifican la familia de direcciones y el tipo de socket. **`AF_INET`** es la familia de direcciones de Internet para IPv4 . **`SOCK_STREAM`** es el tipo de socket para TCP , el protocolo que se utilizará para transportar nuestros mensajes en la red.

`bind()` se utiliza para asociar el socket con una interfaz de red y un número de puerto específicos:

```
HOST = '127.0.0.1'  # Standard loopback interface address (localhost)
PORT = 65432        # Port to listen on (non-privileged ports are > 1023)

# ...

s.bind((HOST, PORT))
```

Los valores pasados **`bind()`**dependen de la familia de direcciones del socket. En este ejemplo, estamos usando **`socket.AF_INET`** (IPv4). Por lo que espera una 2-tupla: **`(host, port)`**.

`host` puede ser un nombre de host, una dirección IP o una cadena vacía. Si se utiliza una dirección IP, `host` debe ser una cadena de direcciones con formato IPv4. La dirección IP `127.0.0.1` es la dirección IPv4 estándar para la interfaz de bucle invertido , por lo que solo los procesos del host podrán conectarse al servidor. Si pasa una cadena vacía, el servidor aceptará conexiones en todas las interfaces IPv4 disponibles.

`port` debe ser un número entero de `1`- `65535`(`0`está reservado). Es el número de puerto TCP para aceptar conexiones de clientes. Algunos sistemas pueden requerir privilegios de superusuario si el puerto es < `1024`.

Aquí hay una nota sobre el uso de nombres de host con **`bind()`**:

> “Si usa un nombre de host en la porción de host de la dirección de socket IPv4 / v6, el programa puede mostrar un comportamiento no determinista, ya que Python usa la primera dirección devuelta por la resolución DNS. La dirección del socket se resolverá de manera diferente en una dirección IPv4 / v6 real, según los resultados de la resolución de DNS y / o la configuración del host. Para un comportamiento determinista, utilice una dirección numérica en la parte del host ". (Fuente)

Discutiré esto más adelante en uso de nombres de host , pero vale la pena mencionarlo aquí. Por ahora, solo comprenda que, al usar un nombre de host, podría ver resultados diferentes dependiendo de lo que devuelva el proceso de resolución de nombres (DNS).

Podría ser cualquier cosa. La primera vez que ejecute su aplicación, podría ser la dirección `10.1.2.3`. La próxima vez que se trata de una dirección diferente, `192.168.0.1`. La tercera vez, podría ser `172.16.7.8`, y así sucesivamente.

Continuando con el ejemplo del servidor, **`listen()`** habilita las **`accept()`** conexiones de un servidor . Lo convierte en un socket de "escucha":

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

`listen()` tiene un `backlog` parámetro. Especifica el número de conexiones no aceptadas que permitirá el sistema antes de rechazar nuevas conexiones. A partir de Python 3.5, es opcional. Si no se especifica, `backlog` se elige un valor predeterminado .

Si su servidor recibe muchas solicitudes de conexión simultáneamente, aumentar el `backlog` valor puede ayudar al establecer la longitud máxima de la cola para conexiones pendientes. El valor máximo depende del sistema. Por ejemplo, en Linux, consulte `/proc/sys/net/core/somaxconn`.

`accept()` bloquea y espera una conexión entrante. Cuando un cliente se conecta, devuelve un nuevo objeto de socket que representa la conexión y una tupla que contiene la dirección del cliente. La tupla contendrá `(host, port)` para conexiones IPv4 o `(host, port, flowinfo, scopeid)` para IPv6.

Una cosa que es imperativo entender es que ahora tenemos un nuevo objeto socket de **`accept()`**. Esto es importante ya que es el socket que usará para comunicarse con el cliente. Es distinto del socket de escucha que usa el servidor para aceptar nuevas conexiones:

```
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data:
            break
        conn.sendall(data)
```

Después de conseguir el objeto socket de cliente **`conn`** a partir de **`accept()`**, un bucle infinito `while` se utiliza para recorrer el bloqueo de llamadas a **`conn.recv()`**. Esto lee los datos que envía el cliente y los repite utilizando **`conn.sendall()`**.

Si **`conn.recv()`** devuelve un objeto vacío de `bytes` `b''`, entonces el cliente cerró la conexión y el bucle finaliza. La `with` instrucción se usa con **`conn`** para cerrar automáticamente el socket al final del bloque.

Echo Client

Ahora echemos un vistazo al cliente `echo-client.py`:

```
#!/usr/bin/env python3

import socket

HOST = '127.0.0.1'  # The server's hostname or IP address
PORT = 65432        # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)

print('Received', repr(data))
```

b'...' literals = a sequence of octets (integers between 0 and 255)

En comparación con el servidor, el cliente es bastante simple. Crea un objeto socket, se conecta al servidor y llama **`s.sendall()`** para enviar su mensaje. Por último, llama **`s.recv()`** para leer la respuesta del servidor y luego la imprime con print.

Running the Echo Client and Server

Ejecutemos el cliente y el servidor para ver cómo se comportan e inspeccionar lo que está sucediendo. Se puede hacer tanto en Windows, como en Linux o Mac. En caso de usar Windows, tenéis que lanzar los scripts desde la terminal de Windows (CMD). No utilicéis el IDLE de Python ya que no funciona.

Abra una terminal o símbolo del sistema, navegue hasta el directorio que contiene sus scripts y ejecute el servidor:

```
$ ./echo-server.py      LINUX/MAC
```

```
> python echo-server.py        WINDOWS CMD
```

Su terminal parecerá colgarse. Eso es porque el servidor está bloqueado (suspendido) en una llamada:

```
conn, addr = s.accept()
```

Está esperando una conexión de cliente. Ahora abra otra ventana de terminal o símbolo del sistema y ejecute el cliente:

```
$ ./echo-client.py     LINUX/MAC
Received b'Hello, world'
```

```
> python echo-server.py        WINDOWS CMD
```

En la ventana del servidor, debería ver:

```
$ ./echo-server.py
Connected by ('127.0.0.1', 64623)
```

En la salida anterior, el servidor imprimió la **`addr`** tupla de la que regresó **`s.accept()`**. Esta es la dirección IP del cliente y el número de puerto TCP. Lo más probable es que el número de puerto sea diferente cuando lo ejecute en su máquina (no tiene por qué salir el `64623` en vuestra máquina también).

Visualización del estado del socket

Para ver el estado actual de los sockets en su host, use **`netstat`**. Está disponible de forma predeterminada en macOS, Linux y Windows.

Aquí está la salida de netstat de macOS después de iniciar el servidor:

```
$ netstat -an
Active Internet connections (including servers)
```

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  127.0.0.1.65432        *.*                    LISTEN
```

Note que `Local Address` es `127.0.0.1.65432`. Si `echo-server.py` hubiera usado en `HOST = ''` en lugar de `HOST = '127.0.0.1'`, netstat mostraría esto:

```
$ netstat -an
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  *.65432                *.*                    LISTEN
```

`Local Address` es `*.65432`, lo que significa que todas las interfaces de host disponibles que admiten la familia de direcciones se utilizarán para aceptar conexiones entrantes. En este ejemplo, en la llamada a **`socket()`**, **`socket.AF_INET`** se utilizó (IPv4). Esto se puede ver en la columna `Proto`: `tcp4`.

Recorté la salida anterior para mostrar solo el servidor de eco. Es probable que vea mucha más salida, dependiendo del sistema en el que lo esté ejecutando. Las cosas para destacar son las columnas `Proto`, `Local Address` y `(state)`. En el último ejemplo anterior, netstat muestra que el servidor de eco está usando un socket TCP IPv4 (`tcp4`), en el puerto 65432 en todas las interfaces (`*.65432`), y está en estado de escucha (`LISTEN`).

Otra forma de ver esto, junto con información adicional útil, es usar `lsof`(lista de archivos abiertos). Está disponible de forma predeterminada en macOS y se puede instalar en Linux usando su administrador de paquetes, si aún no lo está:

```
$ lsof -i -n
COMMAND     PID   USER   FD   TYPE   DEVICE SIZE/OFF NODE NAME
Python    67982 nathan    3u  IPv4 0xecf272      0t0  TCP *:65432 (LISTEN)
```

`Lsof` le da la `COMMAND`, `PID`(identificador de proceso), y `USER`(identificador de usuario) de tomas abiertas de Internet cuando se utiliza con la `-i` opción. Arriba está el proceso del servidor de eco.

`Netstat` y `lsof` tiene muchas opciones disponibles y difieren según el sistema operativo en el que las esté ejecutando. Consulte la `man` página o la

documentación para ambos. Definitivamente vale la pena pasar un poco de tiempo con ellos y conocerlos. Serás recompensado. En macOS y Linux, use man `netstat` y `man lsof`. Para Windows, utilice `netstat /?` O `netstat` help.

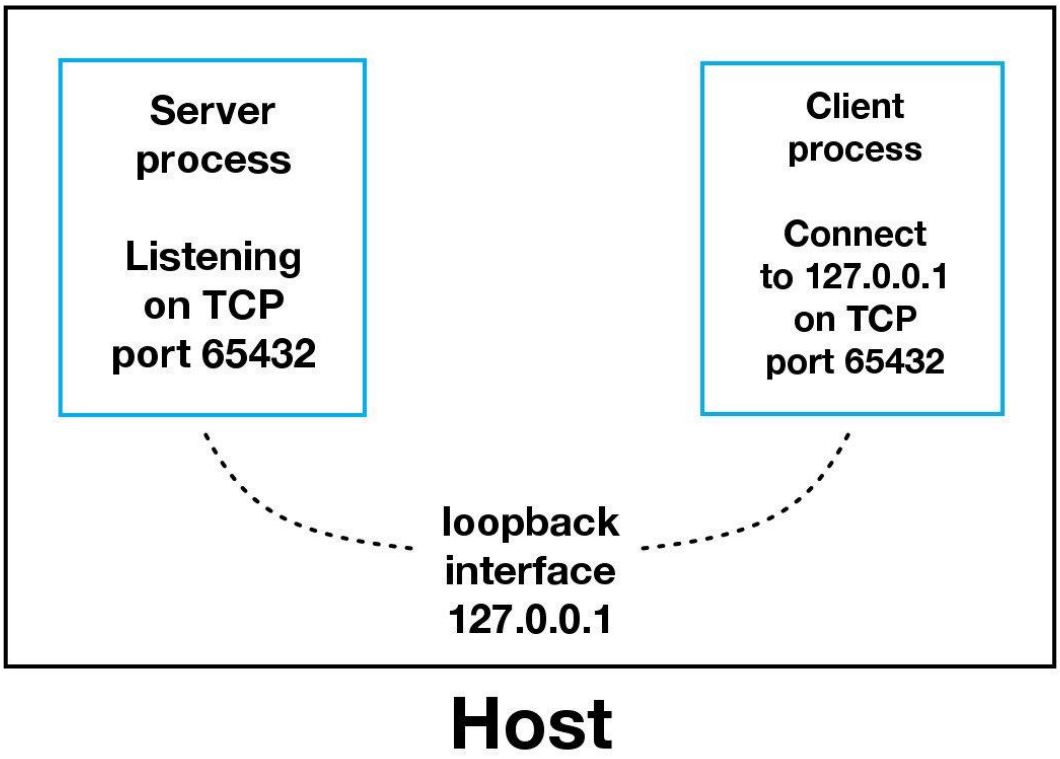
Aquí hay un error común que verá cuando se realiza un intento de conexión a un puerto sin conector de escucha:

```
$ ./echo-client.py
Traceback (most recent call last):
  File "./echo-client.py", line 9, in <module>
    s.connect((HOST, PORT))
ConnectionRefusedError: [Errno 61] Connection refused
```

O el número de puerto especificado es incorrecto o el servidor no se está ejecutando. O tal vez hay un firewall en la ruta que bloquea la conexión, lo que puede ser fácil de olvidar. También puede ver el error `Connection timed out`. Obtenga una regla de firewall agregada que permita al cliente conectarse al puerto TCP.

Fallo de comunicación

Echemos un vistazo más de cerca a cómo el cliente y el servidor se comunicaron entre sí:



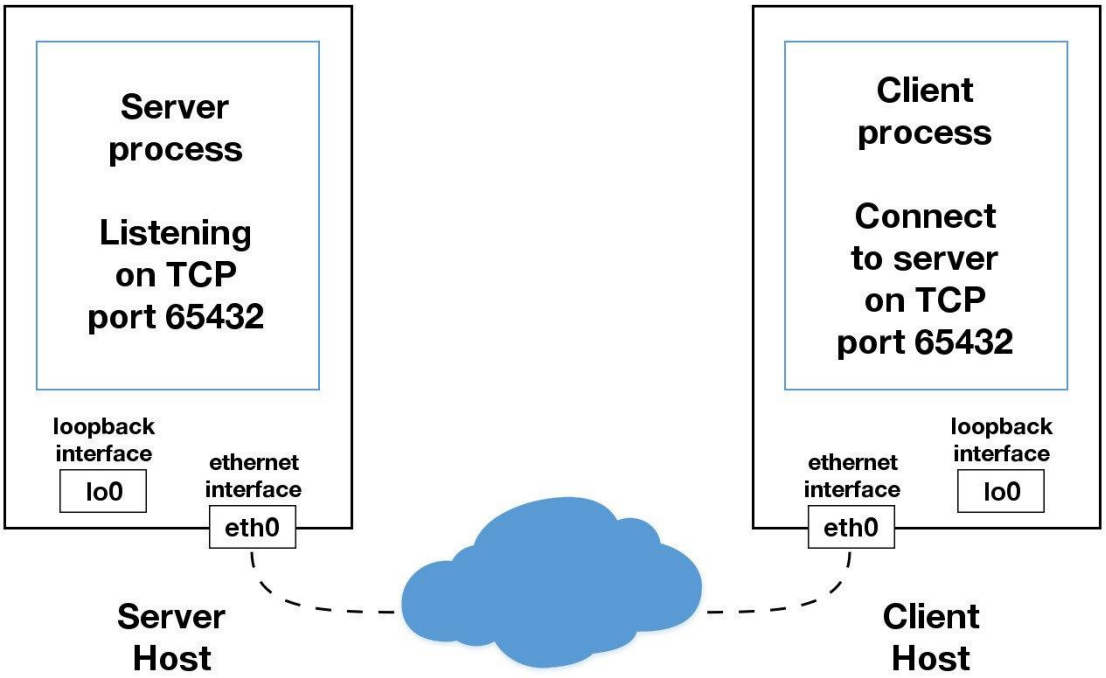
Cuando se utiliza la interfaz de bucle invertido (dirección IPv4 `127.0.0.1`o dirección IPv6 `::1`), los datos nunca abandonan el host ni tocan la red externa. En el diagrama anterior, la interfaz de loopback está contenida dentro del host. Esto representa la naturaleza interna de la interfaz de bucle invertido y que las conexiones y los datos que la transitan son locales para el host. Esta es la razón por la que también escuchará la interfaz de bucle invertido y la dirección IP `127.0.0.1`o se le `::1` denominará "localhost".

Las aplicaciones utilizan la interfaz de bucle invertido para comunicarse con otros procesos que se ejecutan en el host y para la seguridad y el aislamiento de la red externa. Dado que es interno y accesible solo desde el host, no está expuesto.

Puede ver esto en acción si tiene un servidor de aplicaciones que usa su propia base de datos privada. Si no es una base de datos utilizada por otros servidores, probablemente esté configurada para escuchar conexiones solo en la interfaz de bucle invertido. Si este es el caso, otros hosts de la red no se pueden conectar.

Cuando usa una dirección IP que no sea `127.0.0.1`o `::1`en sus aplicaciones, probablemente esté vinculada a una interfaz Ethernet que esté conectada

a una red externa. Esta es su puerta de entrada a otros hosts fuera de su reino "localhost":

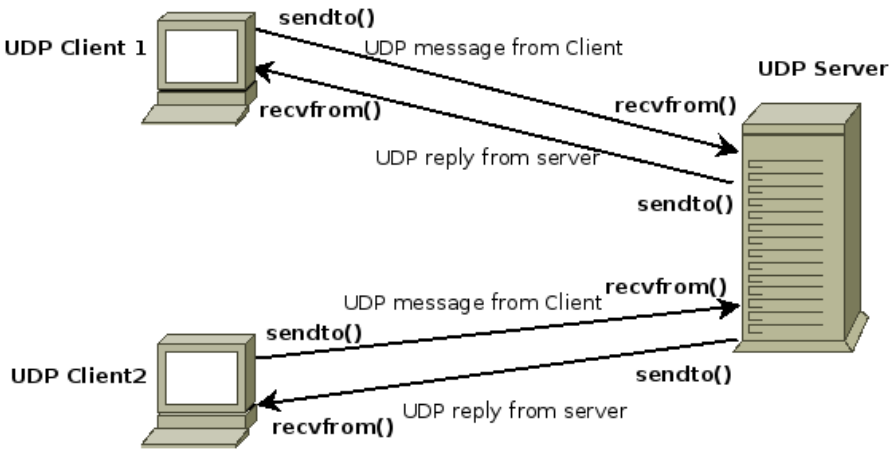


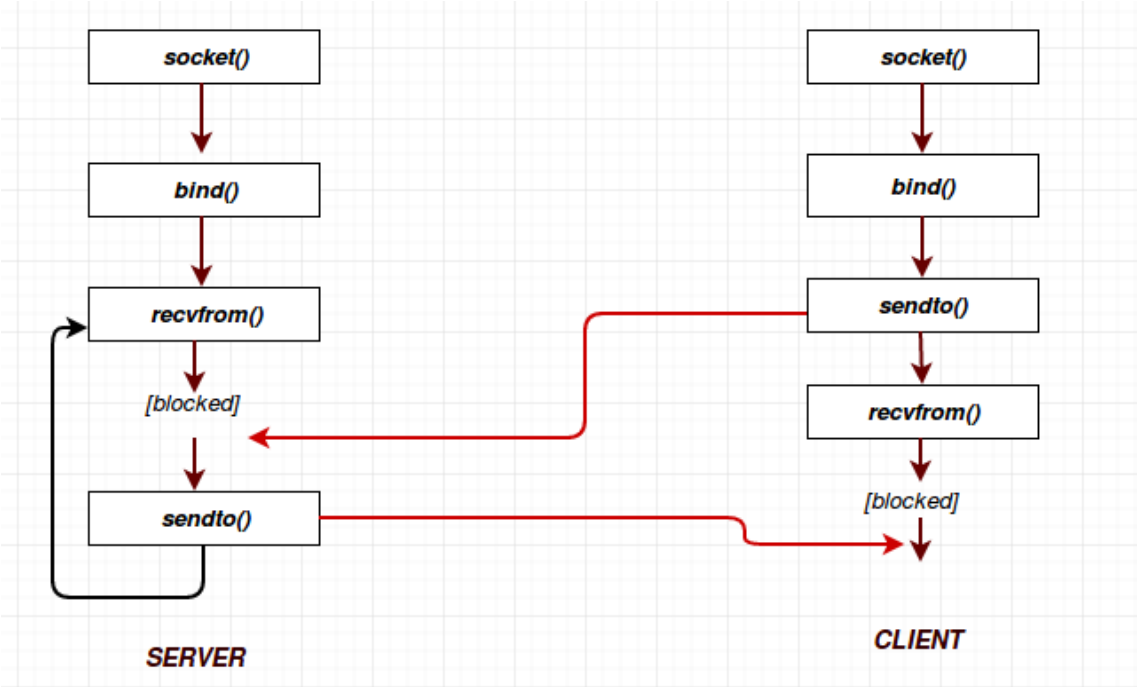
Ten cuidado ahí fuera. Es un mundo cruel y desagradable. Asegúrese de leer la sección Uso de nombres de host antes de aventurarse desde los confines seguros de "localhost". Hay una nota de seguridad que se aplica incluso si no usa nombres de host y solo usa direcciones IP.

Sockets protocolo UDP

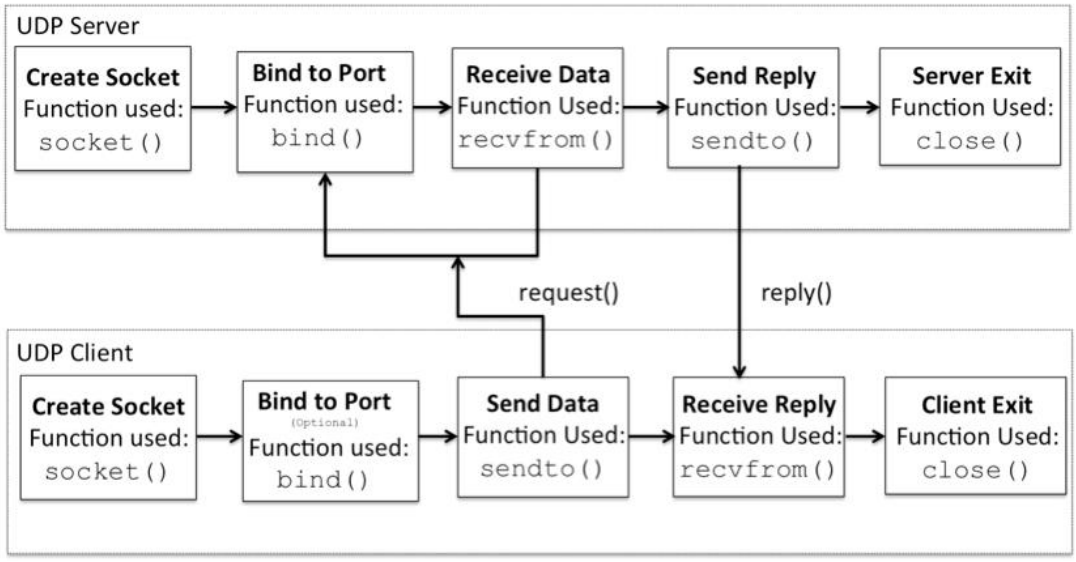
Como hemos visto en clase y en el libro de CCNA, UDP es un protocolo sin conexión, es decir, ya no necesitamos establecer ninguna conexión por lo que no necesitaremos los métodos vistos antes para TCP de socket.listen() ni socket.accept().

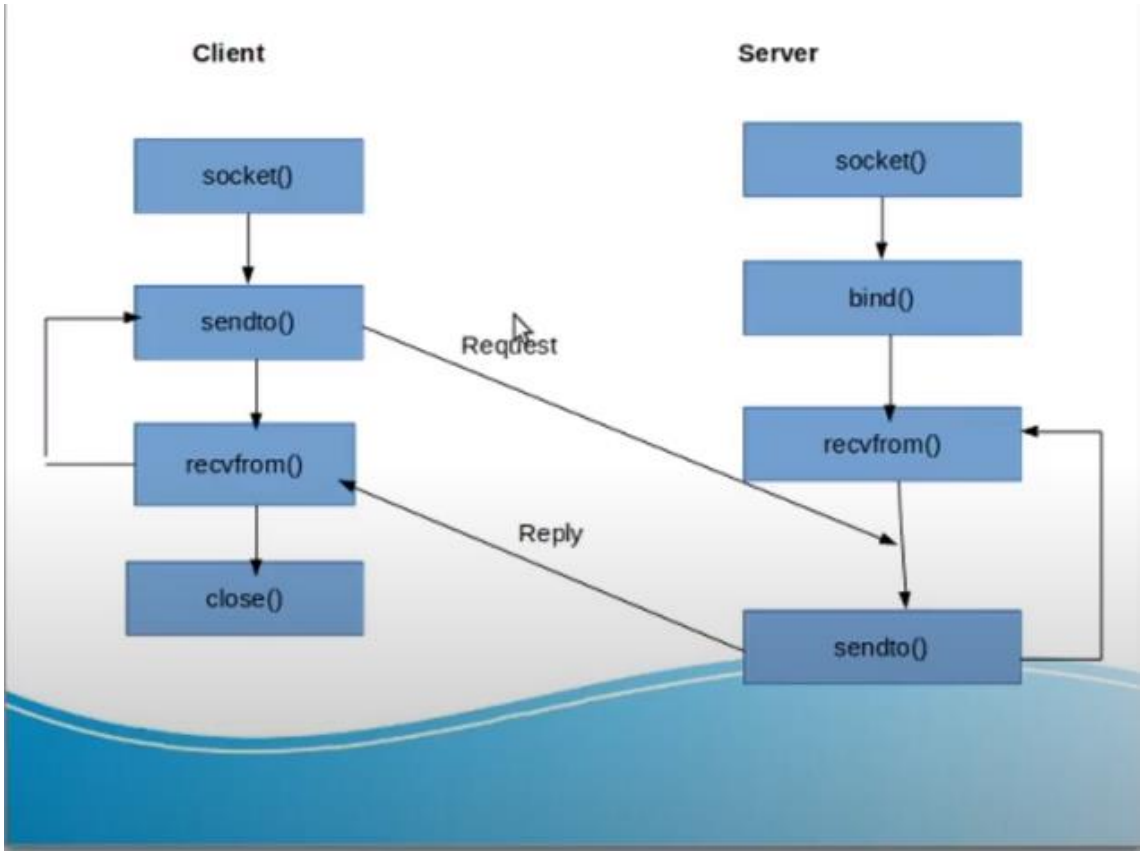
En UDP funciona como podemos ver en las siguientes imágenes el envío de paquetes:





Como vemos, ya no tenemos ni el método listen() ni el accept(). Por parte del servidor solo necesitamos crear el socket, enlazarlo al puerto y la IP y esperar a que llegue algún paquete UDP con el método recvfrom(). Una vez recibe alguún paquete y lo muestra o hace lo que tenga que hacer con el servidor, podemos enviar un paquete de vuelta al cliente con el método sendto().





Ahora veamos el código para crear un echo-server y un echo-client con el protocolo UDP.

Servidor echo en UDP:

```
#!/usr/bin/env python3
import socket

HOST = 'localhost'
PORT = 65400

buffersize = 4096


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

s_sock.bind((HOST,PORT))

while True:

    print("Escuchando peticiones desde", s_sock.getsockname())
    data, addr = s_sock.recvfrom(buffersize)
    print("Mensaje del Cliente:", data.decode('utf-8'))

    s_sock.sendto(data, addr)
```

Cliente echo en UDP:

```
#!/usr/bin/env python3
import socket

HOST = 'localhost'
PORT = 65400
buffersize = 4096

c_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
msg = input(">")
c_sock.sendto(msg.encode('utf-8'),(HOST,PORT))
data, addr = c_sock.recvfrom(4096)
print("Mensaje del Servidor ECHO:", data.decode('utf-8'))
```