Ejercicio Teórico

Contesta las preguntas en azul y rellena los lugares dónde hay puntos.

IMPORTANTE: Utiliza tus palabras, no copies información

1. Explica que es un socket y qué elementos básicos necesita.
   Se trata de una solución en forma de código que sirve para establecer y gestionar la comunicación de información en una red, algo así como un punto de encuentro entre emisor y receptor en una "conversación" (intercambio de datos). Sus elementos básicos son la dirección IP y el número de puerto.

2. Pon un ejemplo detallando los pasos de una comunicación Cliente-Servidor con Stream Socket (orientado a conexión). Para cada función indica qué hace y qué parámetros necesita.

   Ejemplo (ampliando cada punto indicado):

   - **El servidor se prepara:**

   1. El servidor crea un socket con el método socket(socket.AF_INET, socket.SOCK_STREAM)
   2. Luego vincula el socket a una dirección (hostname, número de puerto) con el método bind().
   3. A continuación, el servidor ejecuta el método listen() para escucha las solicitudes de conexión de los cliente que quieran sus servicios.
   4. Cuando un cliente intenta conectarse, el servidor espera y acepta la conexión con la función accept(), que crea un nuevo socket para dicho cliente y devuelve una tupla (cliente, dirección).
   5. Ahora ambos pueden comunicarse. El servidor usará recv() para recibir datos y sendall() para enviarlos.
   6. Finalmente, el servidor cierra la comunicación y el socket con el método close().

   - **El cliente se prepara:**

   1. El cliente crea un socket con el método socket(socket.AF_INET, socket.SOCK_STREAM)
   2. El cliente ejecuta el método connect() indicando el hostname y el número de puerto del servidor al que quiere solicitar una información o un servicio.
   3. Tras recibir el accept() del servidor, el cliente puede empezar a comunicarse con el servidor.
   4. Mediante los métodos sendall() y recv() enviará y recibirá datos al/del servidor.
   5. Finalmente, el cliente cierra el socket para liberar recursos con close().

   - **Comunicación entre el servidor y el cliente:**

   1. El servidor ha recibido la solicitud del cliente y ejecuta un Accept() para vincularse con el cliente. ¿Cómo lo hace? Accept() devuelve una tupla (cliente, dirección). Cliente es el un nuevo socket creado en el servidor para atender las peticiones.
   2. El cliente envía una solicitud al servidor con el método send().
   3. El servidor recibe la solicitud en el socket cliente con el método receive().
   4. Ahora el servidor envía una respuesta con el método send()/sendall(). Ambos tienen como parámetro el mensaje a enviar, pero send() devuelve el número de bytes que se han logrado enviar.
   5. El cliente recibe los datos con el método recv() en el que se debe especificar el tamaño máximo de los datos en bytes (p.e. 1024).

6. El cliente, finalmente cierra su socket con close(), que no requiere parámetros, pero se usa de la siguiente forma: cliente_socket.close() (vamos a llamar así al socket de la comunicación en la parte del cliente).
7. El servidor recibe la petición de cierre con recv().
8. Antes de cerrar el socket, ambos comprueban si quedan datos por enviar en los buffers (y TCP intenta enviarlos antes del cierre).
9. El servidor finalmente cierra su socket también con cliente.close() (llamamos así al socket en la parte del servidor).

3. Pon un ejemplo detallando los pasos de una comunicación Cliente-Servidor con Datagram Socket (no orientado a conexión). Para cada función indica que hace y qué parámetros necesita.

- **El servidor se prepara:**

1. El cliente crea un socket con el método socket(socket.AF_INET, socket.SOCK_DGRAM)
2. El servidor vincula el socket a una dirección (hostname y número de puerto, que también son sus parámetros) con bind().
3. Una vez vinculado, el servidor espera con recvfrom() que tiene como parámetro un número máximo de bytes que recibirá el servidor, y devolverá una tupla con (mensaje y dirección del cliente).
4. Tras recibir los datos, el servidor envía una respuesta con sendto() a la dirección del cliente, sendto() tiene como parámetros la respuesta y la dirección (IP y puerto).
5. Nuevamente, el servidor vuelve a esperar con recvfrom() hasta que el cliente envíe el último mensaje (close()) .

- **El cliente se prepara:**

1. El cliente crea un socket con el método socket(socket.AF_INET, socket.SOCK_DGRAM).
2. El cliente envía una solicitud al servidor con sendto(), el método funciona igual que en el servidor.
3. El cliente espera una respuesta con el método recvfrom(), al igual que en el servidor.
4. Finalmente, el cliente cierra el socket con close().

- **Comunicación entre el servidor y el cliente:**

1. El servidor espera con recvfrom() a recibir un mensaje del cliente y responde con sendto().
2. Al igual que el servidor, el cliente espera con recvfrom() a recibir una respuesta del servidor y envía peticiones con sendto().

4. Enumera las diferencias entre una comunicación Cliente-Servidor con Stream Socket y Datagram Socket.

- **Stream Socket (TCP)**: Es orientado a la conexión. Antes de la transmisión de datos, se establece una conexión entre el cliente y el servidor. Garantiza la entrega fiable y ordenada de los datos. Si hay pérdida de datos, TCP se encarga de reintentarlo hasta que se entreguen correctamente. Implementa control de flujo para evitar overruns y asegurar que el receptor no reciba más datos de los que puede manejar. Debido a su fiabilidad, TCP es más lento, ya que incluye mecanismos para la corrección de errores y el control de flujo. Requiere una fase de "handshake" (3-way-handshake) para establecer la conexión. Consume más recursos del sistema. Todo ello lo hace ideal para aplicaciones donde se requiere confiabilidad, como transferencia de archivos (FTP), navegación web (HTTP), o aplicaciones de chat en tiempo

real.

- **Datagram Socket (UDP)**: No orientado a la conexión. No se establece una conexión antes de la transmisión, los datos se envían directamente. No garantiza la entrega ni el orden de los datos. Los paquetes pueden llegar duplicados, desordenados o no llegar. No ofrece control de flujo, lo que significa que los datos pueden enviarse sin restricciones, independientemente de si el receptor puede manejarlos (puede producir overruns). Es más rápido debido a la falta de mecanismos de fiabilidad y control de flujo. No requiere establecimiento ni cierre de conexión, cada mensaje se envía independientemente. Consume menos recursos. Por ello lo es adecuado para aplicaciones donde la velocidad es crítica y la fiabilidad no es esencial, como la transmisión de vídeo en tiempo real, VoIP o juegos online.

PREGUNTAS DE INVESTIGACIÓN:

5. Explica el proceso de establecimiento de una conexión en el modelo de sockets TCP/IP, desde el envío de una solicitud de conexión hasta que se establece la comunicación entre el cliente y el servidor. Describe los pasos del protocolo de enlace de tres vías (three-way handshake) y cómo se implementa en Python, añade un ejemplo de código.

   El proceso de establecimiento de una conexión en el modelo de sockets TCP/IP consta de 4 pasos:

1. **Envío de SYN por parte del cliente**:
   El cliente crea un socket TCP y envía una solicitud de conexión con el flag SYN activado (synchronize). Esto indica al servidor que el cliente desea iniciar una conexión, e incluye un número de secuencia inicial (ISN por sus siglas en inglés). Aquí comienza el ***3-ways-hadshake***.

2. **El servidor Responde con SYN-ACK**:
   El servidor crea un socket para comunicarse con el cliente con accept() y manda un paquete con los flags SYN y ACK activados(synchronize-acknowledgment, algo así como “acuerdo/reconocimiento de sincronización”) para responder a la solicitud aceptando. Además, el servidor también incluye su número de secuencia inicial (ISN) y el número de confirmación será el número de secuencia del cliente +1).

3. **Confirmación del cliente (ACK):**
   Tras recibir el paquete SYN-ACK del servidor, el cliente confirma la conexión con un paquete con el flag ACK activado. El ISN del cliente se incrementa en 1, y el número de confirmación será el de secuencia del servidor +1. Una vez finalizado el ***3-ways-hadshake*** la conexión está establecida y ambos pueden comunicarse.

   Nota: Al final del documento están los códigos para copiar-pegar.

Ejemplo de código:

VSC - Python > examples > 3-ways-handshake > 3wh-cliente.py > ...

```python
import socket

"""
3-ways-handshake
CÓDIGO DEL CLIENTE

"""

# Crear un socket TCP
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Conectar al servidor
client_socket.connect(('localhost', 12345))

# Enviar datos al servidor
client_socket.sendall("¡Hola, servidor!".encode())

# Recibir una respuesta del servidor
data = client_socket.recv(1024)
print(f"Recibido: {data.decode()}")

# Cerrar la conexión
client_socket.close()
```

Img. 1 Código cliente

VSC - Python > examples > 3-ways-handshake > 3wh-servidor.py > ...

```python
import socket
"""
3-ways-handshake
CÓDIGO DEL SERVIDOR

"""
# Crear un socket TCP
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Vincular el socket al puerto y la dirección
server_socket.bind(('localhost', 12345))

# Escuchar las solicitudes de conexión
server_socket.listen(1)
print("Esperando conexión...")

# Aceptar una conexión entrante
client_socket, client_address = server_socket.accept()
print(f"Conexión establecida con {client_address}")

# Recibir datos del cliente
data = client_socket.recv(1024)
print(f"Recibido: {data.decode()}")

# Enviar una respuesta al cliente
client_socket.sendall("¡Hola, cliente! Conexión establecida.".encode())

# Cerrar la conexión
client_socket.close()
server_socket.close()
```

Img. 2 Código servidor

6. Describe cómo se pueden manejar múltiples conexiones de clientes en un servidor de Python usando sockets. Compara el uso de threading vs asyncio para esta tarea, incluyendo ventajas y desventajas de cada enfoque en términos de rendimiento y escalabilidad.

   Manejar múltiples conexiones de clientes en un servidor de Python implica que el servidor debe poder comunicarse con varios clientes simultáneamente. Esto se puede lograr mediante multithreading (hilos) o programación asíncrona (asyncio).

1. **Uso de threading:**
   En este enfoque, se utiliza un hilo separado para gestionar cada conexión con un cliente. Esto significa que el servidor puede manejar múltiples conexiones "en paralelo" desde el punto de vista del usuario.

   Ventajas:
   - Facilidad de implementación
   - Cada conexión se maneja de forma independiente.
   - Compatibilidad con bibliotecas de terceros.
   - Maneja de conexiones concurrentes adecuado con baja carga de clientes.

   Desventajas:
   - Crear un hilo por conexión es costoso en términos de memoria y CPU. No escala bien con un nivel elevado de conexiones (+100 o +1000).
   - El GIL (Global Interpreter Lock) impide que varios hilos ejecuten bytecode de Python en paralelo. Esto limita la capacidad de aprovechar múltiples núcleos de CPU.
   - Si los hilos comparten datos, es necesario implementar mecanismos de sincronización (como locks), lo que puede ser propenso a errores y complicar el código.

2. **Uso de asyncio:**
   Asyncio permite manejar múltiples conexiones mediante un bucle de eventos asíncrono,

evitando el bloqueo y los costos de los hilos. Este enfoque utiliza co-rutinas en lugar de hilos.

Ventajas:
- Puesto que maneja miles de conexiones concurrentes con menor consumo de memoria y CPU que los hilos, ya que no necesita crear uno por cliente, es más escalable.
- Mientras una co-rutina espera, el bucle de eventos puede ejecutar otras tareas.
- No se necesitan locks porque todo corre en un único hilo del sistema.

Desventajas:
Curva de aprendizaje: Puede ser menos intuitivo para principiantes debido a la necesidad de entender corutinas, async y await.
- No todas las bibliotecas de terceros son compatibles con asyncio.
- Monohilo: Aunque escala bien con conexiones, asyncio no aprovecha múltiples núcleos de CPU por sí solo.

ANEXO:

Nota: A continuación los códigos de las capturas img. 1 e img. 2:

Img. 1:

```
import socket

"""
3-ways-handshake
CÓDIGO DEL CLIENTE

"""

# Crear un socket TCP
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Conectar al servidor
client_socket.connect(('localhost', 12345))

# Enviar datos al servidor
client_socket.sendall("¡Hola, servidor!".encode())

# Recibir una respuesta del servidor
data = client_socket.recv(1024)
print(f"Recibido: {data.decode()}")

# Cerrar la conexión
client_socket.close()
```

Img. 2:

```python
import socket
"""
3-ways-handshake
CÓDIGO DEL SERVIDOR

"""
# Crear un socket TCP
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Vincular el socket al puerto y la dirección
server_socket.bind(('localhost', 12345))

# Escuchar las solicitudes de conexión
server_socket.listen(1)
print("Esperando conexión...")

# Aceptar una conexión entrante
client_socket, client_address = server_socket.accept()
print(f"Conexión establecida con {client_address}")

# Recibir datos del cliente
data = client_socket.recv(1024)
print(f"Recibido: {data.decode()}")

# Enviar una respuesta al cliente
client_socket.sendall("¡Hola, cliente! Conexión establecida.".encode())

# Cerrar la conexión
client_socket.close()
server_socket.close()
```