GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

Unidad 2.3

Programación de comunicaciones en red

Índice

Sumario

Licencia

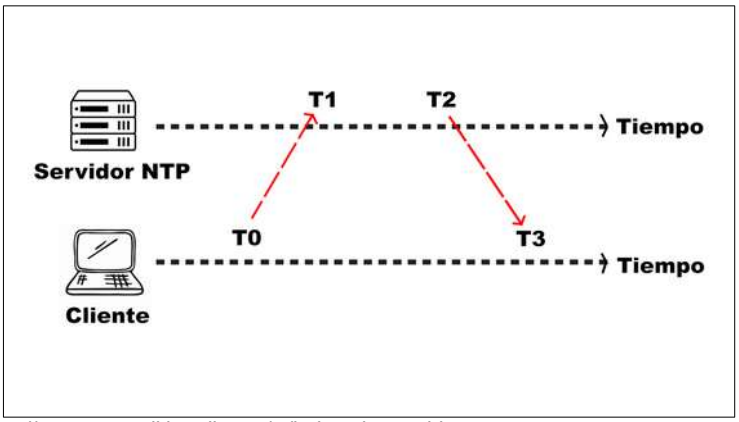
1. Comunicación Cliente-Servidor

1.1. Respuesta unidireccional

En una configuración de cliente-servidor con una única respuesta, el cliente inicia una conexión con el servidor para solicitar un servicio o información específica. En este escenario, el servidor responde a la solicitud del cliente proporcionando la información requerida y, posteriormente, cierra la conexión. Un ejemplo común es el servicio de obtener la hora de un servidor o un servidor web que nos devuelve una página web.

Se ha trabajado este modelo en la unidad anterior. En dicha ocasión se presentó un cliente que conecta con un servidor hora utilizando una conexión de tipo UDP. En esta unidad vamos a trabajar este modelo de comportamiento con una conexión TCP.

Dado el cliente os propongo hacer la parte de servidor. Debe poder gestionar varios clientes a la vez. (Ver tareas evaluables).



Autor: RaulSantosRecio - https://commons.wikimedia.org/w/index.php?curid=118061937

Comportamiento de cada parte:

- **Servidor:**
  - Escucha.
  - Recibe una petición TCP de un cliente.
  - Mira la hora del sistema.
  - Contesta con la hora al cliente.
  - Cierra la conexión.
- **Cliente:**
  - Se conecta al servidor.
  - Espera que le responda con la hora.
  - Imprime la hora por pantalla.
  - Cierra la conexión.

Nota: Asegúrate de ejecutar primero el servidor y luego el cliente.

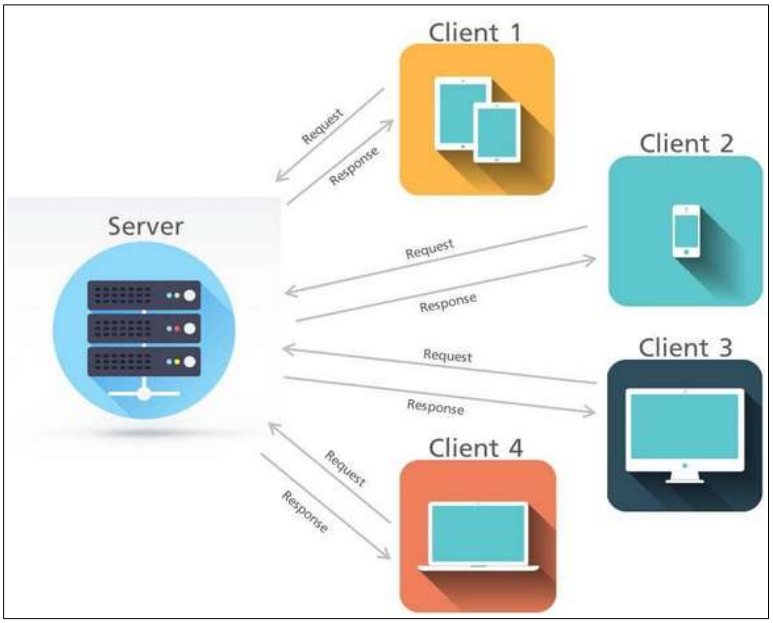
1.2. Conversación entre cliente y servidor

En este modelo se define la relación entre dos programas, el cliente y el servidor, que interactúan para realizar una tarea o servicio específico. En esta configuración, el servidor, que actúa como proveedor de recursos o servicios, utiliza el programa cliente para interactuar con el usuario.

The Sergio Caredda Blog by Sergio Caredda -https://sergiocaredda.eu

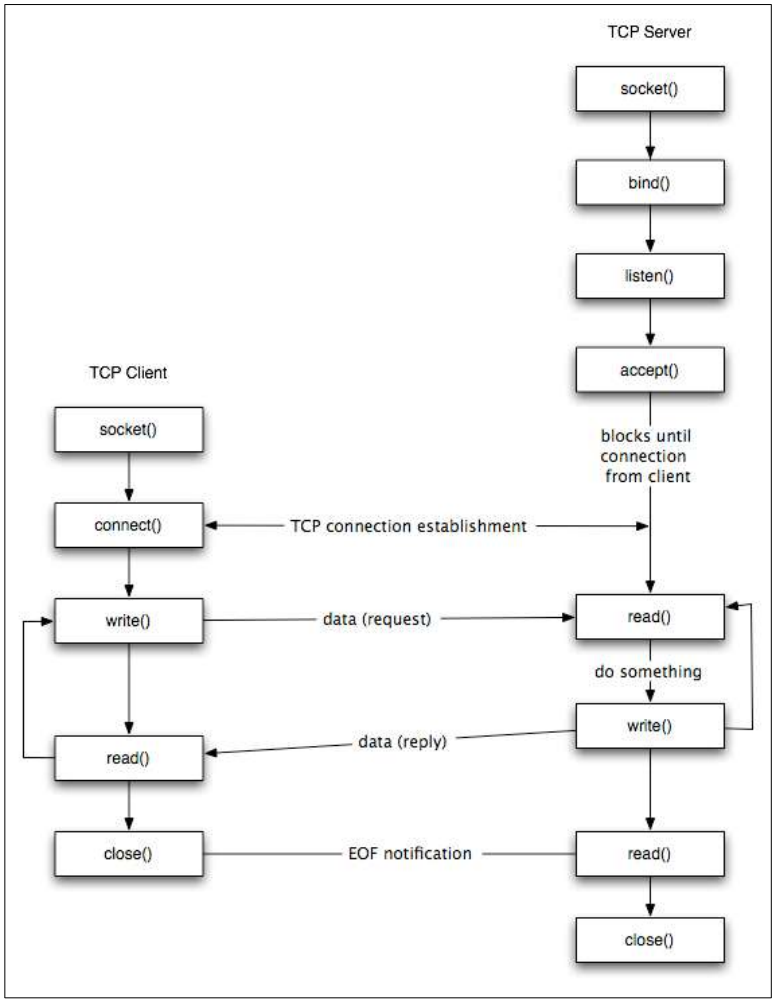
La implementación de este modelo cliente-servidor implica la creación de programas que gestionan la comunicación bidireccional, como el envío y recepción de mensajes. Este intercambio de información entre el cliente y el servidor se podría considerar como una conversación simulada.

Este enfoque es fundamental en el diseño de sistemas distribuidos y aplicaciones en red, permitiendo la modularidad, escalabilidad y eficiencia en el intercambio de información entre diferentes componentes.



Autora: Elena Spadini - https://elespdn.github.io

El diagrama de secuencia que seguiría este modelo sería el siguiente:



https://stackoverflow.com/questions/40046981/java-tcp-socket-can-the-server-make-the-first-request-first

Dificultades de este modelo - bloqueo

La comunicación entre cliente y servidor presenta una complejidad intrínseca, donde el riesgo de desincronización surge al enviar múltiples mensajes. En este escenario, existe la posibilidad de que el receptor lea varios mensajes en una única instrucción de **recv()**, lo que puede conducir a una falta de sincronización entre las partes.

Esta desincronización, al no interpretar adecuadamente los límites de cada mensaje, puede dar lugar a bloqueos tanto en el servidor como en el cliente. Para evitar este problema, es esencial implementar mecanismos de control de flujo y protocolos de comunicación que permitan una interpretación clara y precisa de los mensajes enviados y recibidos.

La dinámica de comunicación entre el cliente y el servidor puede compararse con un juego de ping pong, donde ambos participantes intercambian mensajes de manera secuencial. Inicia con el servidor enviando datos utilizando la función "send", seguido por el cliente que los recibe mediante "recv". Posteriormente, el cliente toma la iniciativa, enviando su propio mensaje a través de "send", y el servidor lo recibe con "recv".

Este intercambio continuo, similar a los vaivenes de una partida de ping pong, refleja la naturaleza interactiva y bidireccional de la comunicación en el modelo cliente-servidor.

- Servidor send() - Cliente recv()
- Cliente send() - Servidor recv()
- Servidor send() - Cliente recv()
- Cliente send() - Servidor recv()
- Servidor send() - Cliente recv()
- Cliente send() - Servidor recv()

En este ejemplo se muestra un código que no funciona. Parecería que funciona, pero tanto el servidor como el cliente se quedan bloqueados:

Servidor:	Cliente:

mensaje = "Bienvenido/a al juego”
self.socket.send(mensaje.encode()) # envía mensaje bienvenida
	data = socket_cliente.recv(1024) #El servidor nos da la bienvenida
	print(data.decode())

Mensaje = “Dime tu nombre por favor: "
self.socket.send(mensaje.encode())
	data = socket_cliente.recv(1024) #El servidor pide el nombre
	print(data.decode())

Name = self.socket.recv(1024).decode() # recibe el nombre self.nombreJugador = name #actualizo la variable del constructor de current thread al nombre del jugador	name = input("> ") #El jugador no da su nombre socket_cliente.sendall(name.encode()) #Enviamos el nombre del jugador
mensaje = f"Hola {name}\n" self.socket.send(mensaje.encode()) # envía mensaje hola y nombre cliente	data = socket_cliente.recv(1024) print(data.decode()) #El servidor dice Hola al jugador

Porqué se queda bloqueado:

El servidor:

- send(Bienvenida)

- send(Tu nombre?)

- nombre=recv() **espera** que el cliente le envié el nombre

El cliente:

- recv(1024) # Recibe:**BienvenidaTu nombre?**. Lo junta todo en un recv()

- recv() **espera** que el servidor le envíe más cosas (Tu nombre?)

A continuación se ofrecen algunas soluciones para evitar el bloqueo:

a) Juntar los mensajes.

En vez de hacer dos send, se junta toda la información en un único send.

mensaje = "Bienvenido/a al juego: piedra, papel o tijera.\nDime tu nombre por favor: "

Servidor:	Cliente:

mensaje = "Bienvenido/a al juego: piedra, papel o
mensaje = "Bienvenido/a al juego"
self.socket.send(mensaje.encode()) # envía
	data = socket_cliente.recv(1024) #El servidor nos da la bienvenida y pide el nombre
	print(data.decode())

Name = self.socket.recv(1024).decode() # recibe el nombre	name = input("> ") #El jugador no da su nombre socket_cliente.sendall(name.encode()) #Enviamos el nombre del jugador
self.nombreJugador = name #actualizo la variable del constructor de current thread al nombre del jugador	
mensaje = f"Hola {name}\n" self.socket.send(mensaje.encode()) # envía mensaje hola y nombre cliente	data = socket_cliente.recv(1024) print(data.decode()) #El servidor dice Hola al jugador

Esta solución tiene un problema si no controlas bien el cuerpo de los bucles. Puede que al final del bucle se haga un send y al inicio del bucle otro send. En este caso, también son dos send seguidos (consecutivos):

```
while not salir:
    mensaje = "Menú de opciones"
    self.socket.send(mensaje.encode())

    mensaje = "Muy bien sigue jugando"
    self.socket.send(mensaje.encode())
```

b) Usar un protocolo de comunicación en el cual confirmo cada envío.

Utilizar un mensaje como por ejemplo “OK” para confirmar el mensaje y romper así los dos send() seguidos.

Servidor:	Cliente: mensajeOK = "OK"

mensaje = "Bienvenido/a al juego: piedra, papel o tijera.”
self.socket.**send**(mensaje.encode()) # envía mensaje bienvenida
	data = socket_cliente.**recv**(1024) #El servidor nos da la bienvenida
	print(data.decode())

sincroOK = self.socket.recv(1024) # leo OK	**socket_cliente.sendall(mensajeOK.encode())** #para sincronizar (envío la pelota de vuelta para que me envíe más cosas)

Mensaje = “Dime tu nombre por favor: "
self.socket.**send**(mensaje.encode())
	data = socket_cliente.**recv**(1024) #El servidor pide el nombre
	print(data.decode())

Name = self.socket.**recv**(1024).decode() # recibe el nombre self.nombreJugador = name #actualizo la variable del constructor de current thread al nombre del jugador	name = input("> ") #El jugador no da su nombre socket_cliente.**sendall**(name.encode()) #Enviamos el nombre del jugador
mensaje = f"Hola {name}\n" self.socket.**send**(mensaje.encode()) # envía mensaje hola y nombre cliente	data = socket_cliente.**recv**(1024) print(data.decode()) #El servidor dice Hola al jugador

Explicación más detallada de esta solución.

Utilizando el ejemplo anterior de programa que se queda bloqueado:

El servidor:

- **send**(Bienvenida)
- **send**(Tu nombre?)
- nombre=recv() **espera** que el cliente le envié el nombre

El cliente:

- recv(1024) # Recibe:**BienvenidaTu nombre?**. Lo junta todo en un recv()
- recv() **espera** que el servidor le envíe más cosas. (Tu nombre?)

Se podría solucionar el problema, como indica este apartado, utilizando un protocolo de comunicación que verifique cada envío:

El servidor:

- **send**(Bienvenida)
- mensaje_sinc = **recv**(1024) #OK
- **send**(Tu nombre?)
- nombre=recv()

El cliente:

- recv(1024) # Recibe: Bienvenida
- send(ok) # protocolo de sincronización
- recv(1024) # Recibe: Tu nombre?
- nombreJugador=input(Tu nombre?)
- send(nombreJugador)

En este ejemplo, se ha usado el protocolo de sincronización únicamente cuando era necesario, si teníamos un mensaje que enviar, se enviaba el mensaje. Se podría implementar, por ejemplo, este protocolo para cada envío.

2. Guardar información de la partida

En un entorno donde el programa servidor debe gestionar múltiples jugadores simultáneamente, es esencial adoptar estrategias que permitan un manejo eficiente de la información de cada partida.

En este contexto, el uso de **variables globales** para almacenar datos específicos de cada juego se vuelve problemático, ya que no se puede prever cuántos jugadores se conectarán simultáneamente. Las variables globales son más adecuadas para almacenar estadísticas generales del servidor, como el número total de jugadores conectados en un día o la duración promedio de las partidas.

Para abordar la necesidad de gestionar la información individual de cada partida, se recomienda hacerlo de dos formas:

2.1. Uso de Paso Parámetros

El paso de parámetros en funciones específicas permite transmitir la información necesaria para cada jugada de manera directa y controlada. Este método garantiza que la información esté localizada y asociada con la ejecución de funciones específicas, evitando posibles conflictos en entornos multijugador.

Si quisiéramos guardar para cada partida el nombre del jugador o jugadora, lo podríamos programar:

Jugador o jugadora 1:

```
jugador = "María"
nuevo_hilo = threading.Thread(target=hilo_partida, args=(jugador,))
```

Jugador o jugadora 2:

```
jugador = "Pedro"
nuevo_hilo = threading.Thread(target=hilo_partida, args=(jugador,))
```

2.2. Uso de Herencia de Clase

El uso de herencia de clase proporciona una estructura organizada para representar cada partida como una instancia única de una clase. Cada instancia puede tener sus propios atributos, como el nombre del jugador, puntos de la jugada, etc. Este enfoque facilita la administración individual de cada partida y ofrece una solución más escalable y estructurada, especialmente en situaciones donde la complejidad de la información de la partida puede aumentar.

Siguiendo con el mismo ejemplo del punto 2.1, dónde queremos guardar el nombre del jugador o jugadora, lo podríamos programar de la siguiente manera:

```
class hilo_Partida(threading.Thread):
    def __init__(self, nombre_jugador):
        super().__init__()
        self.nombre_jugador = nombre_jugador
        # Otros atributos de la partida
```

Jugador o jugadora 1:

```
partida_jugador = hilo_Partida("María")
```

Jugador o jugadora 2:

```
partida_jugador = hilo_Partida("Pedro")
```