Informe Teórico
“Socket y RPC”

Integrante.
Matías Jesús Egaña Alfaro.

Asignatura
Sistemas Distribuidos.

Profesor
Juan Prudencio Torres Ossandon.

Introducción **3**
Teoría **3**
Socket 3
Modelo Cliente-Servidor 3
RPC 4
Primera actividad (Sockets) 6
Ejecución 7
Segunda Actividad (RPC) 8
Ejecución 10
Conclusiones **11**

Introducción

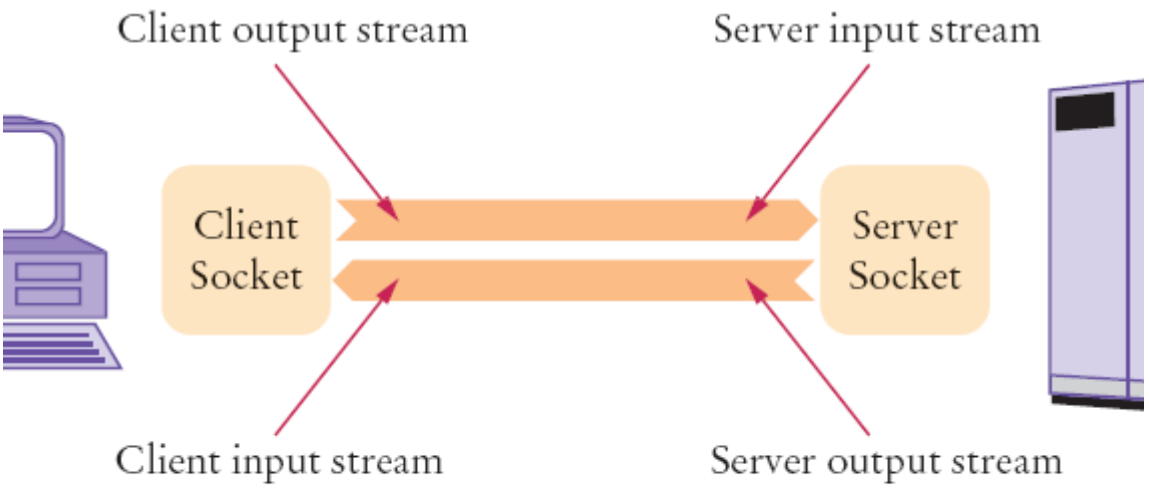
En este presente informe tiene como objetivo dar una explicación sobre el funcionamiento y entendimiento de los software sobre Sockets(Cliente - Servidor) y Rpc(Cliente - Servidor).Perteneciente a los laboratorios y código proporcionado por el profesor de la asignatura.

Teoría

Antes de todo explicaremos brevemente algunos conceptos, que no serán de utilidad más adelante y es importante conocer su concepto.

Socket

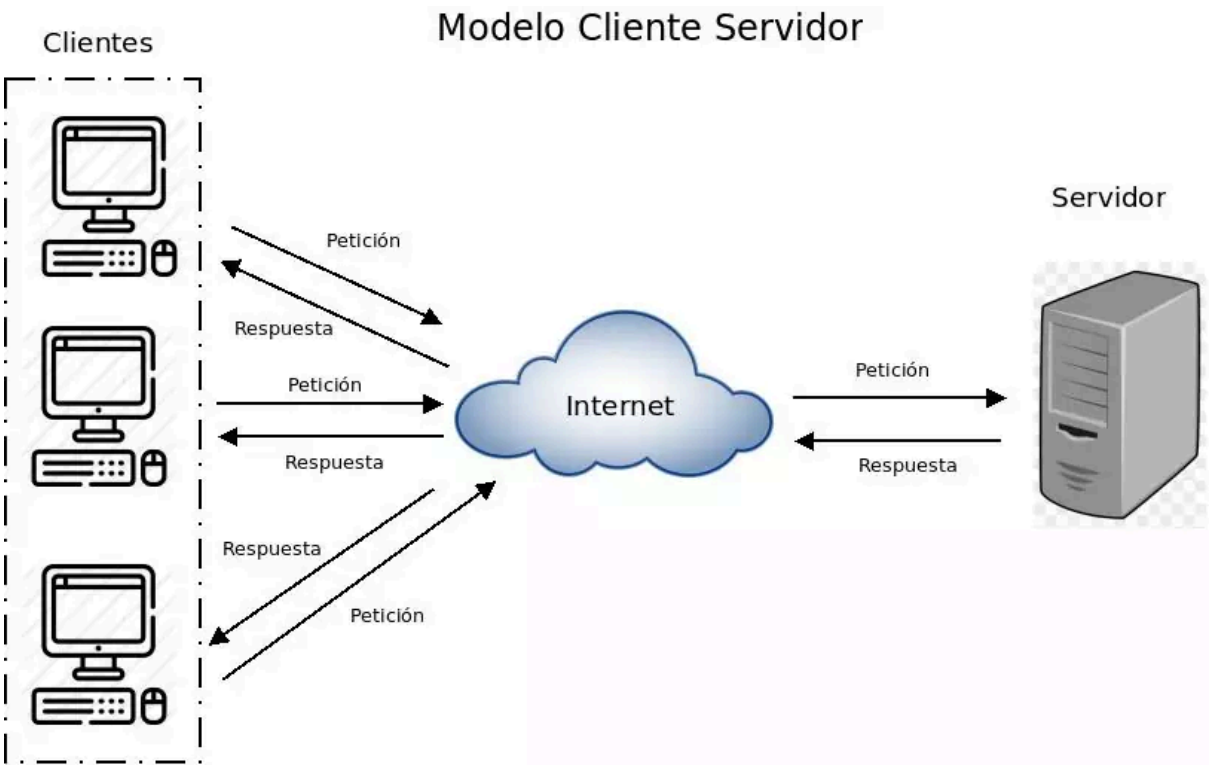
Es un punto final de un enlace de comunicación bidireccional entre dos programas que se ejecutan en la red. Esta permite que los programas se comuniquen entre sí, ya sea en la misma computadora o en computadoras diferentes a través de una red.



“Diagrama funcionamiento de comunicación por sockets”.

Modelo Cliente-Servidor

Es una forma de organizar la interacción entre aplicaciones distribuidas, donde los clientes solicitan servicios o recursos a servidores que los proporcionan. Esta arquitectura es ampliamente utilizada en una variedad de aplicaciones, incluidas las aplicaciones web, las aplicaciones móviles y los sistemas empresariales.



“Diagrama arquitectura Cliente-Servidor”.

RPC

RPC significa "Remote Procedure Call" (Llamada a Procedimiento Remoto) y es un mecanismo que permite que un programa solicite la ejecución de un procedimiento (o función) en otro espacio de memoria, generalmente en una máquina remota en una red.

En sistemas distribuidos, donde los recursos y datos pueden estar distribuidos en varias máquinas, RPC facilita la comunicación entre los diferentes componentes del sistema, permitiendo que un programa solicite la ejecución de un procedimiento en una máquina remota como si estuviera llamando a una función local.

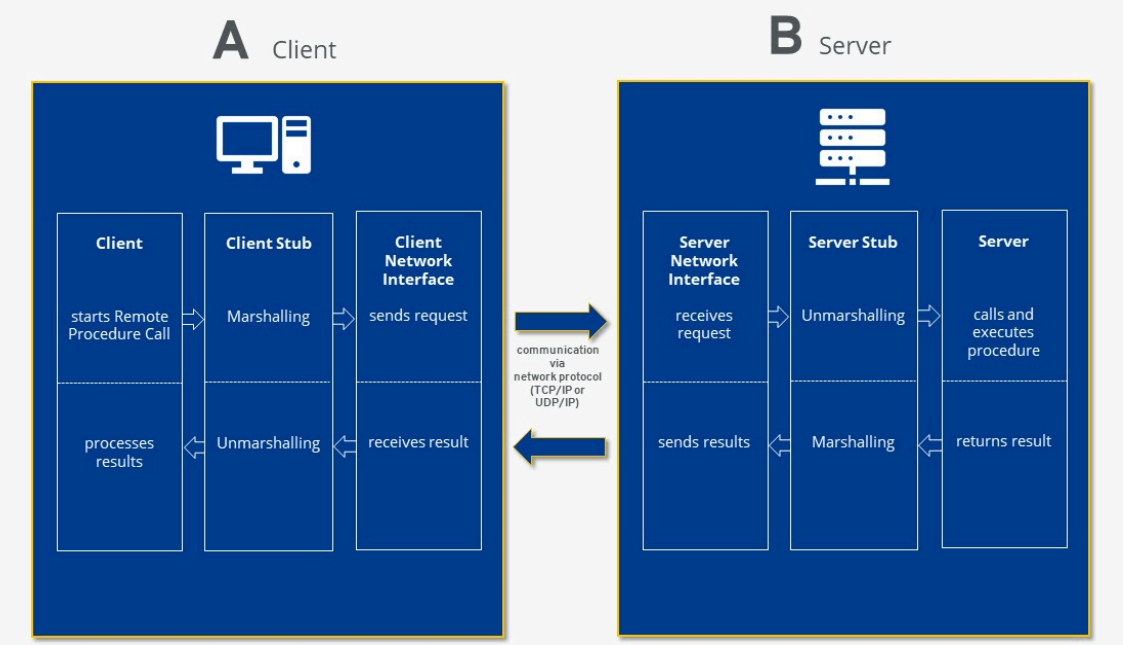
El proceso típico en una llamada a procedimiento remoto implica la serialización de los parámetros de la llamada, la transmisión de estos datos a través de la red, la ejecución del procedimiento remoto en el servidor, la serialización de los resultados (si los hay) y la transmisión de vuelta al cliente.

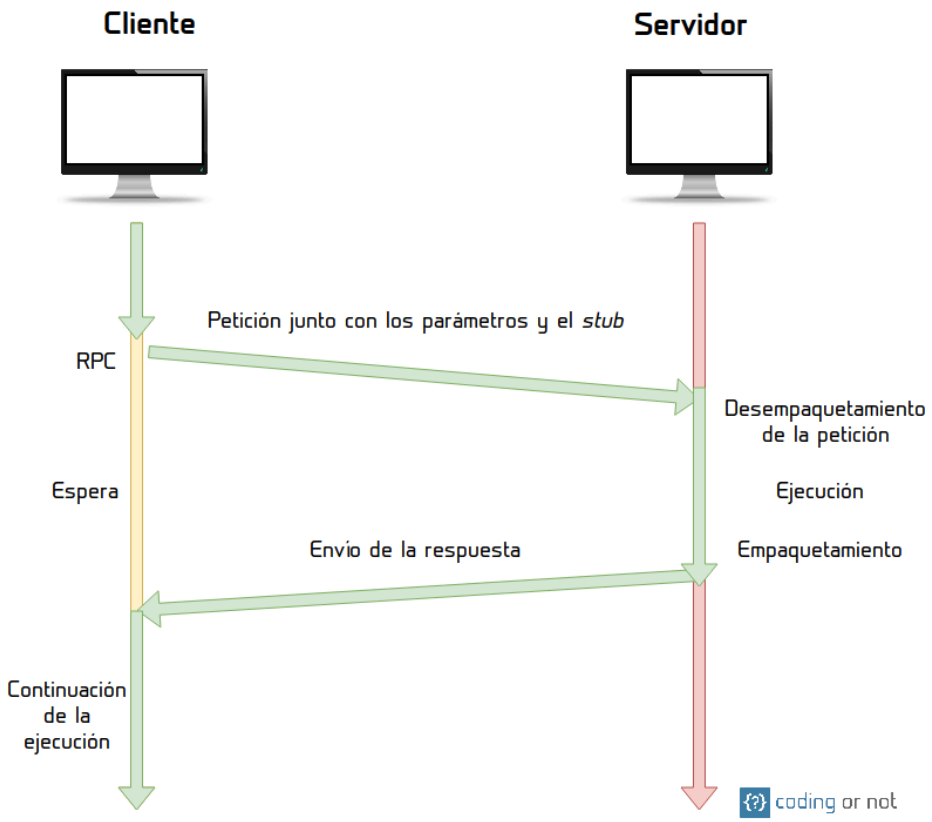
Esta abstracción hace que la comunicación entre procesos distribuidos sea más transparente para el desarrollador, simplificando el desarrollo de sistemas distribuidos.

Cuando un programa desea llamar a un procedimiento en una máquina remota a través de RPC, utiliza el stub[1] correspondiente en su propio espacio de direcciones.

[1] Es una pieza de código que actúa como una interfaz local para un procedimiento remoto.

El stub se encarga de empaquetar los parámetros de la llamada en un formato adecuado para la transmisión a través de la red y enviarlos al servidor remoto que aloja el procedimiento deseado. Una vez que el servidor recibe los datos, el stub del servidor los desempaqueta, invoca el procedimiento real y devuelve los resultados (si los hay) al stub del cliente. Luego, el stub del cliente desempaqueta los resultados y los entrega al programa cliente.





"Diagrama RPC".

Primera actividad (Sockets)

En esta ejecutamos el código en python´s: Socket_cliente.py y Socket_servidor.py

En ambos códigos hay que cambiarle la dirección ipv4. En mi caso seria la conexión **“192.168.1.136”**, para poder establecer la conexión entre el cliente y el servidor por medio de la red wifi.

Socket > Socket_servidor.py > ...

```python
import socket
s = socket.socket()

#Invoco  al metodo bind, pasando como parametro (IP,puerto)
s.bind(("192.168.1.136",  9999))
#Invoco  el metodo listen para escuchar conexiones
#con el numero maximo de  conexiones como parametro
print ("Esperando a los clientes....")
s.listen(5)
#El  metodo accept bloquea la ejecucion a la espera de conexiones
#accept  devuelve un objeto socket y una tupla Ip y puerto
sc, addr = s.accept()
print ("Recibo conexion de " + str(addr[0]) + ":" + str(addr[1])+ " ...escriba Exit para terminar")
while True:
    #invoco  recv sobre el socket cliente, para recibir un maximo de 1024 bytes
    try: recibido = sc.recv(1024)
    except:
        break
    aux=recibido.decode('utf-8')
    #Envio  la respuesta al socket cliente
    try: sc.send(recibido)
    except:
        break
    print (aux)
print  ("...Cliente desconectado")
#cierro  sockets cliente y servidor
sc.close()
s.close()
```

“Codigo “Socket_servidor.py”.

Este código lo que hace es dar la IPv4 y el puerto, da por sockets un acceso máximo de 5 dispositivos conectados.

El método accept() acepta una conexión entrante. Retorna un nuevo socket (sc) que representa la conexión con el cliente y una tupla con la dirección IPv4 y el puerto del cliente.

Luego en el bucle se encarga de procesar los datos si es necesario y luego los envía de vuelta al cliente utilizando el método send(). El tamaño máximo de los datos recibidos en cada llamada a recv() es de 1024 bytes.

Ahora explicaremos la parte del cliente.

```python
import socket
s = socket.socket()
#invoco  el metodo connect del socket pasando como parametro (IP , puerto)
s.connect(("192.168.1.136",  9999))
while  True:
    mensaje=input("> ").encode('utf-8')
    s.send(mensaje)
    #invoco  el metodo send pasando como parametro el string ingresado por el  usuario
    aux=mensaje.decode()
    if  aux.lower() == "exit":
        break
#cierro  socket
s.close()
```

"Codigo "Socket_cliente.py".

El cliente se conecta al servidor en la dirección IPv4: 192.168.1.136 en el puerto 9999.

En este, el cliente entra en un bucle infinito donde espera la entrada del usuario. El mensaje ingresado por el usuario se convierte en bytes usando encode('utf-8') y se envía al servidor usando send(). Si el usuario ingresa "exit", el bucle se rompe y el cliente termina.

Ejecución

Primero ejecutamos el servidor.

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\mati_\OneDrive\Documentos\Proyectos\JT> & C:/Users/mati_/AppData/Local/Programs/Python/Python312/pytho
n.exe c:/Users/mati_/OneDrive/Documentos/Proyectos/JT/Socket/Socket_servidor.py
Esperando a los clientes....
```

"Terminal "Socket_servidor.py".

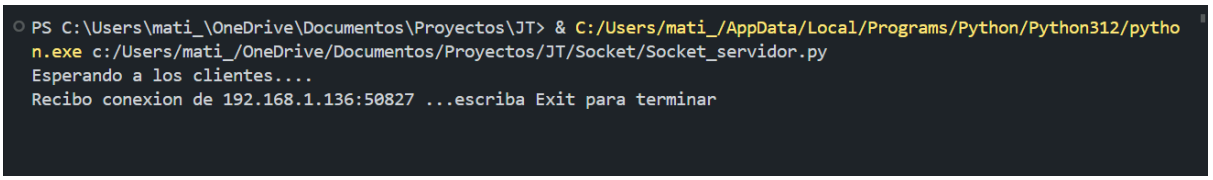
Luego ejecutamos el cliente.

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\mati_\OneDrive\Documentos\Proyectos\JT> & C:/Users/mati_/AppData/Local/Programs/Python/Python312/pytho
n.exe c:/Users/mati_/OneDrive/Documentos/Proyectos/JT/Socket/Socket_cliente.py
>
```

"Terminal "Socket_cliente.py".

Ahora si nos fijamos en la terminal del servidor, veremos un pequeño cambio. Donde se es recibida la conexión

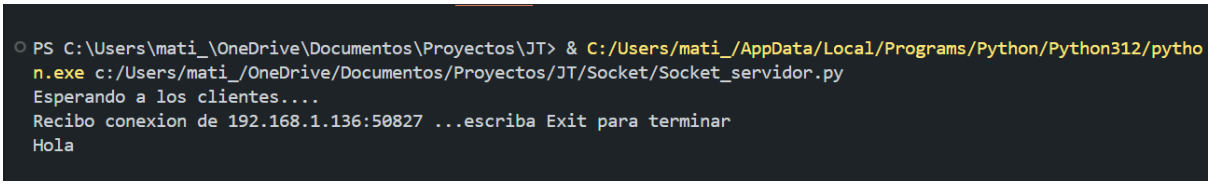


"Terminal "Socket_servidor.py".

Y ahora si mandamos un texto por el cliente, el servidor la recibirá.

Por ejemplo, mandaremos por parte del cliente un mensaje por consolar: "Hola".

"Terminal "Socket_cliente.py".



"Terminal "Socket_servidor.py".

Veremos que el mensaje fue recibido por el servidor y nos muestra por pantalla el mensaje.

Segunda Actividad (RPC)

Este servidor ofrece una función remota simple a través de XML-RPC que determina si un número dado es par o impar.

Utiliza el localhost, que es el servidor para realizar pruebas por defecto y se le asigna el puerto 8000.

Los clientes pueden conectarse a este servidor y llamar a la función is_even para realizar esta verificación.

El servidor se inicia y comienza a esperar conexiones entrantes, manejando las solicitudes de los clientes de forma indefinida.

rpc_cliente.py rpc_server.py

RPC > rpc_server.py > ...

```python
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    if n % 2 == 0:
        return ("Par")
    else:
        return ("Impar")

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

"Codigo "rpc_server.py".

Una vez explicado el servidor, procederemos a explicar el cliente.

Se establece una conexión con el servidor que está escuchando en **http://localhost:8000/**.

El cliente entra en un bucle mientras el número ingresado sea mayor o igual a cero. Dentro del bucle, solicita al usuario que ingrese un número positivo. Si el número es positivo (numero > 0), se llama a la función remota is_even en el servidor a través del proxy y se imprime la respuesta.

rpc_cliente.py rpc_server.py

RPC > rpc_cliente.py > ...

```python
import xmlrpc.client
numero=0
with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:

    while (numero>=0):
        numero=int(input("Ingrese un Número positivo:"))
        if numero >0:
            respuesta=str(proxy.is_even(numero))
            print(numero," es ",respuesta,"\n")

    print("\nFin !!")

```

"Codigo "rpc_cliente.py".

Ejecución

Primero ejecutamos el servidor, como en la actividad anterior.

"Terminal "rpc_server.py".

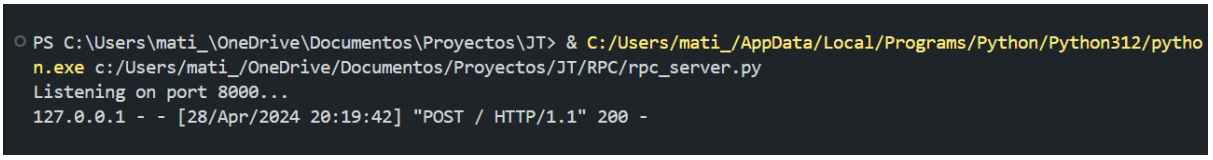
Ahora se encuentra escuchando por parte del puerto, una conexión.

Luego ejecutamos el cliente.

"Terminal "rpc_cliente.py".

Donde nos pedirá un número positivo y para hacer la petición al servidor, nosotros utilizaremos como ejemplo el número 2.

Recibe la petición el servidor.



"Terminal "rpc_server.py".

Para finalmente retornar la respuesta del servidor al cliente, que es que el número ingresado es par.

"Terminal "rpc_cliente.py".

Conclusiones

Estas son algunas conclusiones:

Los códigos demuestran cómo implementar la comunicación entre un cliente y un servidor utilizando sockets TCP y XML-RPC en Python. Ambos protocolos permiten la transferencia de datos entre procesos en diferentes dispositivos a través de una red.

A pesar de que los códigos sean simples, ilustran conceptos importantes sobre la creación de servidores y clientes, así como el registro de funciones remotas en el caso de RPC.

Ofreciendo una introducción práctica a la comunicación cliente-servidor, pero hay mucho más por explorar en este campo, para seguir aprendiendo sobre temas como: arquitecturas distribuidas, servicios web, mensajería asincrónica, entre otros.

Por mi parte esta fue una experiencia enriquecedora para explorar de manera práctica los conceptos vistos en clase, como en asignaturas anteriores.