PROGRAMACION DISTRIBUIDA CON PYTHON

IRVYN XICALE CABRERA -- 201963582

```
  File "C:\Users\irvyn\OneDrive\Documents\programDistribuida\socket\client.py", line 10, in <module>
    s.connect((host,port))
ConnectionRefusedError: [WinError 10061] No se puede establecer una conexión ya que el equipo de destino de
negó expresamente dicha conexión

(escuela) C:\Users\irvyn\OneDrive\Documents\programDistribuida\socket>python server.py
Connected with[addr],[port]('192.168.100.14', 63617)
```

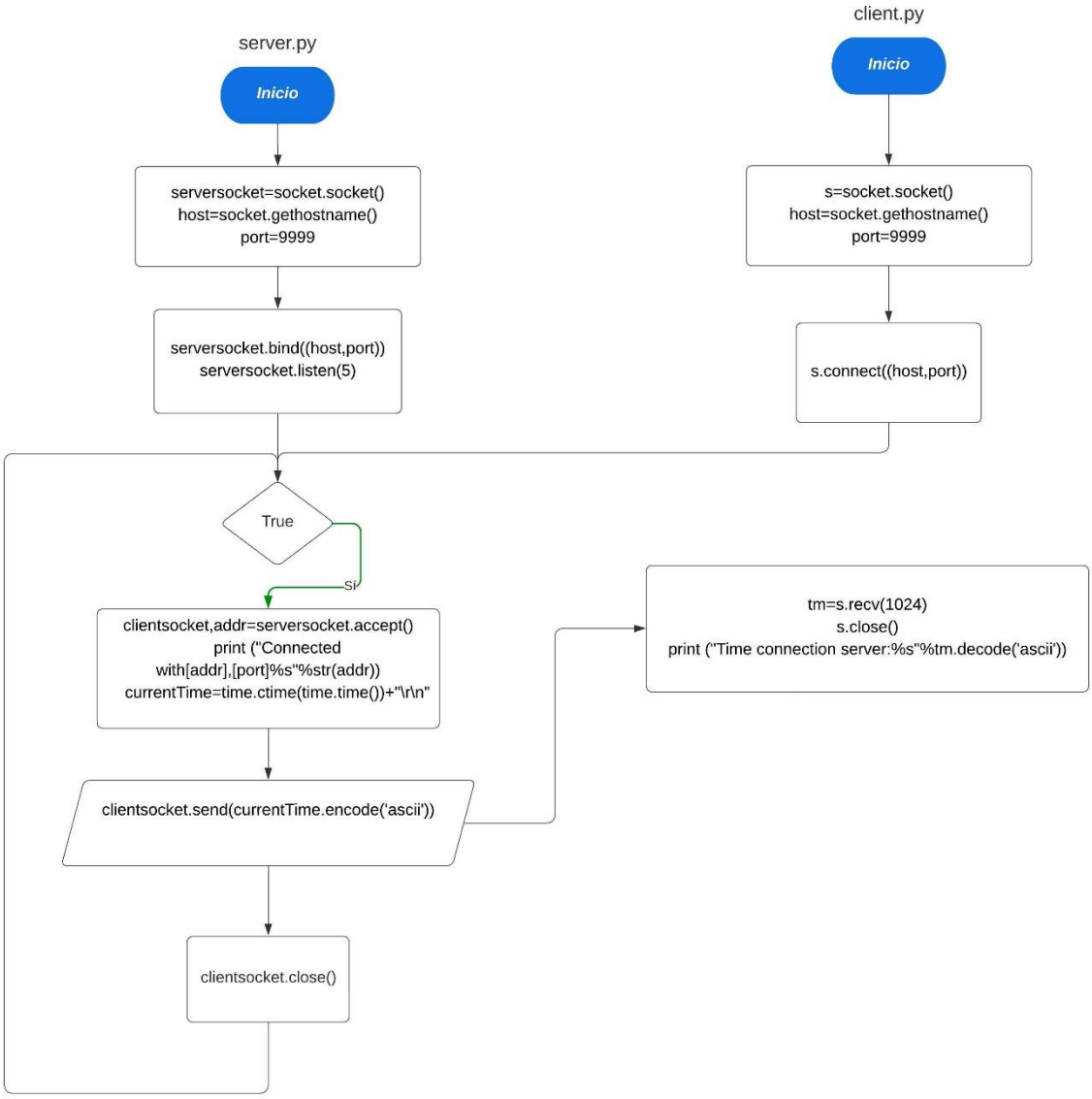
python
cmd

```
(escuela) C:\Users\irvyn\OneDrive\Documents\programDistribuida>cd socket

(escuela) C:\Users\irvyn\OneDrive\Documents\programDistribuida\socket>python client.py
Time connection server:Thu Nov  9 00:55:56 2023


(escuela) C:\Users\irvyn\OneDrive\Documents\programDistribuida\socket>
```

python
cmd

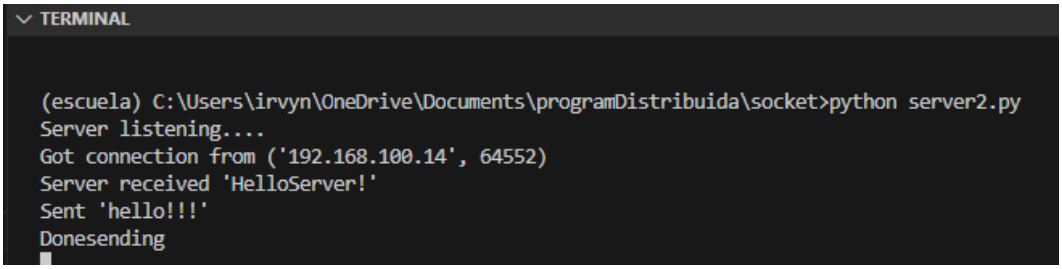


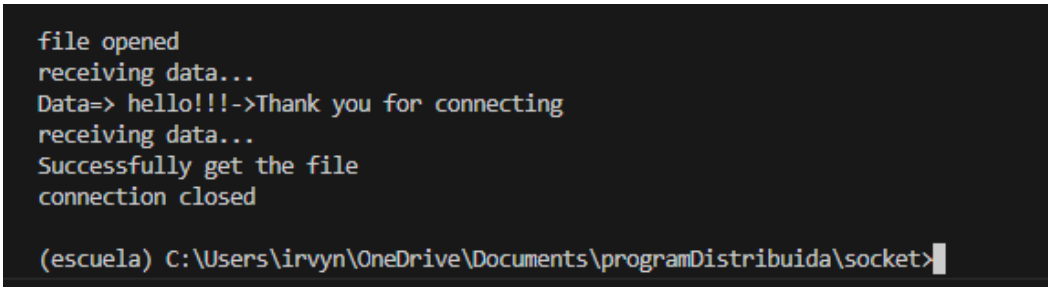
Proceso de conexión del servidor al cliente:

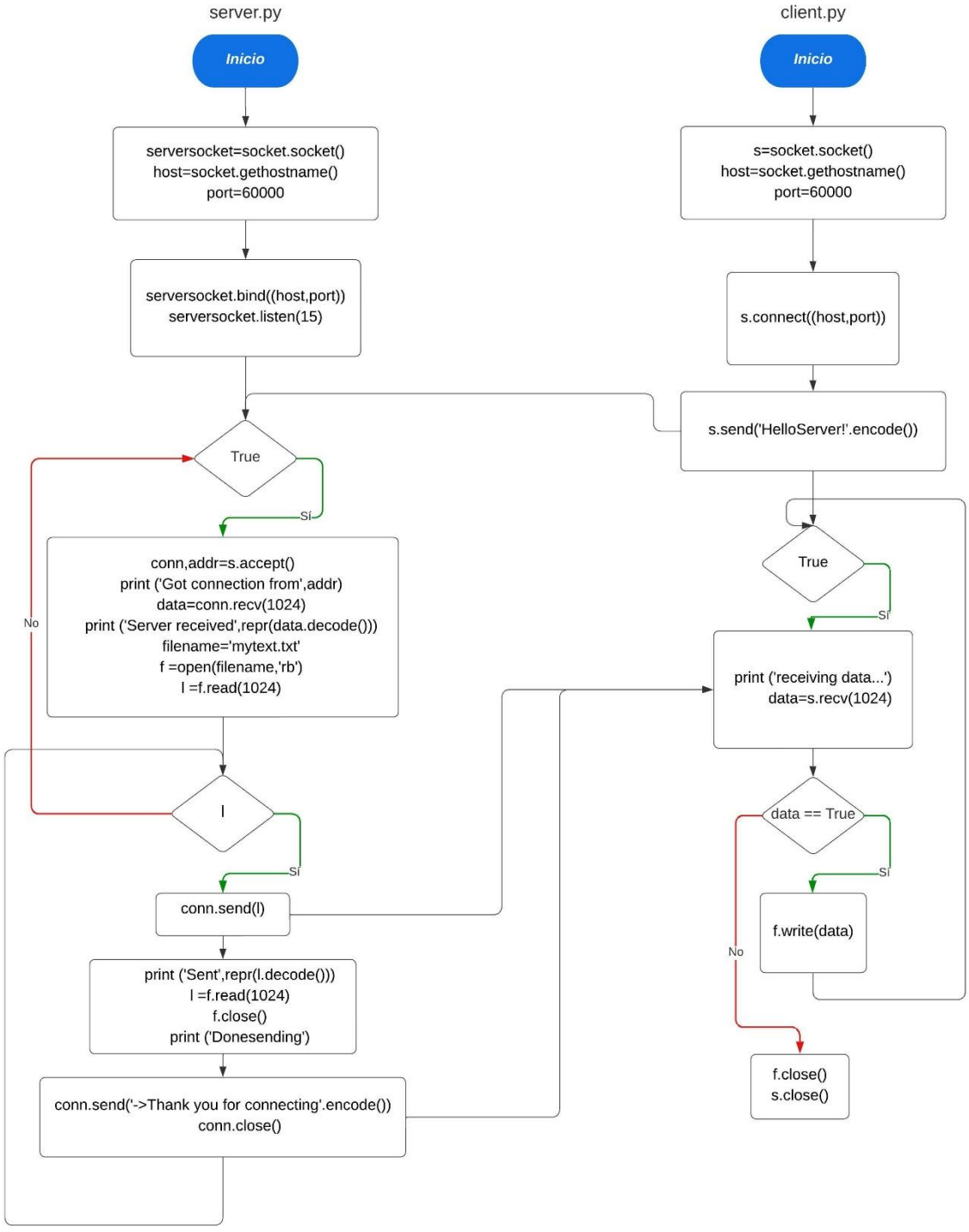
- El servidor inicia creando un objeto de socket (serversocket) con la familia de direcciones AF_INET y el tipo de socket SOCK_STREAM. Este socket se utiliza para aceptar conexiones entrantes de los clientes.
- Luego, el servidor obtiene el nombre de la máquina local mediante socket.gethostname() y almacena ese nombre en la variable "host".
- Se establece el número de puerto en 9999, que es el mismo número de puerto al que el cliente intentará conectarse.
- El servidor utiliza la función bind() para vincular el socket al host y al número de puerto. Esto indica al servidor en qué dirección y puerto debe escuchar las solicitudes entrantes de los clientes.
- Después de realizar el "bind", el servidor utiliza la función listen() para configurar el socket para escuchar hasta 5 solicitudes de conexión en la cola de espera.
- A continuación, el servidor entra en un bucle infinito, esperando continuamente nuevas solicitudes de conexión entrantes.
- Cuando llega una solicitud de conexión entrante desde un cliente, el servidor acepta la conexión utilizando la función accept(). Esto crea un nuevo socket (clientsocket) que se utilizará para comunicarse con el cliente en particular, y también obtiene la dirección del cliente (addr).
- El servidor puede entonces enviar y recibir datos desde el cliente utilizando el clientsocket, en este caso, envía la hora actual al cliente en formato ASCII.
- Después de completar la comunicación con el cliente actual, el servidor cierra la conexión utilizando clientsocket.close().
- El servidor vuelve a su estado de escucha, esperando nuevas solicitudes de conexión entrantes en el bucle infinito.

Proceso de conexión del cliente al servidor:

- El cliente inicia creando un objeto de socket (socket) usando la familia de direcciones AF_INET y el tipo de socket SOCK_STREAM. Este socket se utiliza para establecer una conexión con el servidor.
- Luego, el cliente obtiene el nombre de la máquina local mediante socket.gethostname() y almacena ese nombre en la variable "host".
- Se establece el número de puerto al que el cliente intentará conectarse en la variable "port", que se fija en 9999.
- El cliente utiliza la función connect() para intentar establecer una conexión con el servidor. Utiliza el nombre del host y el número de puerto para conectarse al servidor.
- Si la conexión se establece con éxito, el cliente está listo para enviar y recibir datos desde el servidor.


TERMINAL
(escuela) C:\Users\irvyn\OneDrive\Documents\programDistribuida\socket>python server2.py
Server listening....
Got connection from ('192.168.100.14', 64552)
Server received 'HelloServer!'
Sent 'hello!!!'
Donesending


file opened
receiving data...
Data=> hello!!!->Thank you for connecting
receiving data...
Successfully get the file
connection closed
(escuela) C:\Users\irvyn\OneDrive\Documents\programDistribuida\socket>



Proceso del Servidor (server.py):

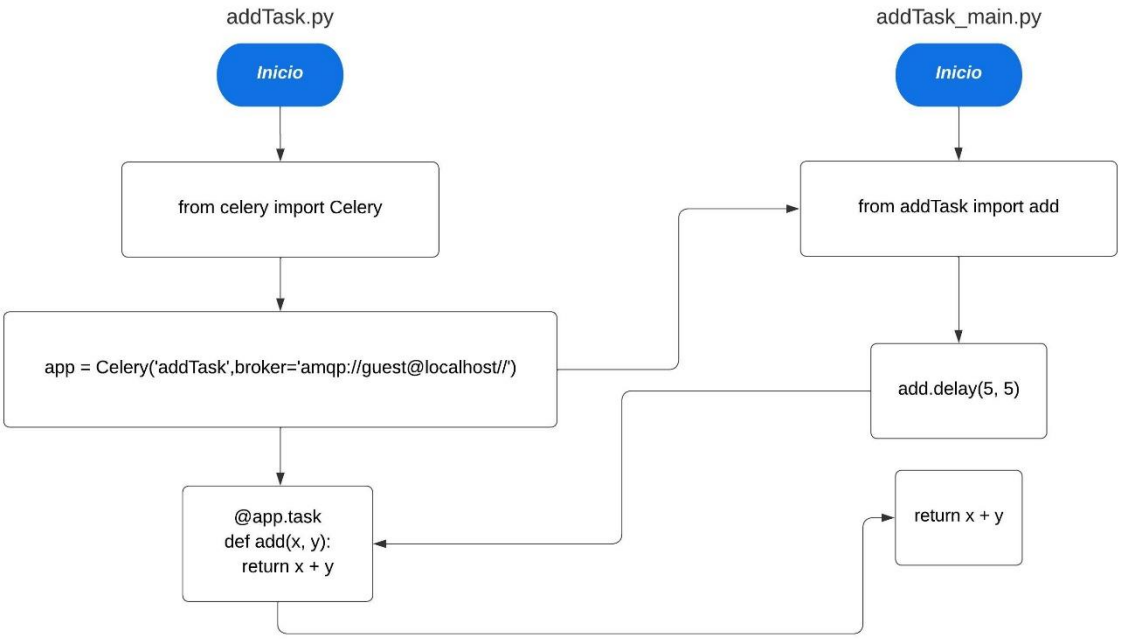
- El servidor comienza creando un objeto de socket utilizando la función socket(). Luego, se obtiene el nombre de la máquina local (host) y se establece el número de puerto en 60000.

- El servidor vincula (bind) el socket a la dirección del host y el puerto especificado utilizando la función bind().
- Luego, el servidor configura el socket para escuchar hasta 15 solicitudes de conexión en la cola de espera utilizando listen().
- El servidor imprime un mensaje indicando que está escuchando en el puerto especificado.
- El servidor entra en un bucle infinito con while True: para esperar continuamente conexiones entrantes.
- Cuando llega una solicitud de conexión desde un cliente, el servidor acepta la conexión utilizando accept(). Esto crea un nuevo socket llamado conn para comunicarse con el cliente y obtiene la dirección del cliente (addr).
- El servidor imprime un mensaje que indica que se ha establecido una conexión con la dirección del cliente.
- El servidor recibe datos del cliente utilizando conn.recv(1024). Esto permite que el servidor reciba hasta 1024 bytes de datos desde el cliente.
- A continuación, el servidor define un nombre de archivo llamado 'mytext.txt' y abre este archivo en modo binario ('rb') utilizando open(). El archivo contiene los datos que se transferirán al cliente.
- El servidor lee 1024 bytes de datos del archivo 'mytext.txt' y los almacena en la variable l.
- Luego, el servidor envía los datos leídos a través del socket conn al cliente utilizando conn.send(l). Esto se repite hasta que ya no hay más datos por enviar.
- El servidor imprime un mensaje indicando que se han enviado los datos.
- Una vez que se han enviado todos los datos del archivo, el servidor cierra el archivo utilizando f.close().
- El servidor envía un mensaje de agradecimiento al cliente utilizando conn.send('->Thank you for connecting'.encode()).
- Finalmente, el servidor cierra la conexión con el cliente utilizando conn.close().

Proceso del Cliente (client2.py):

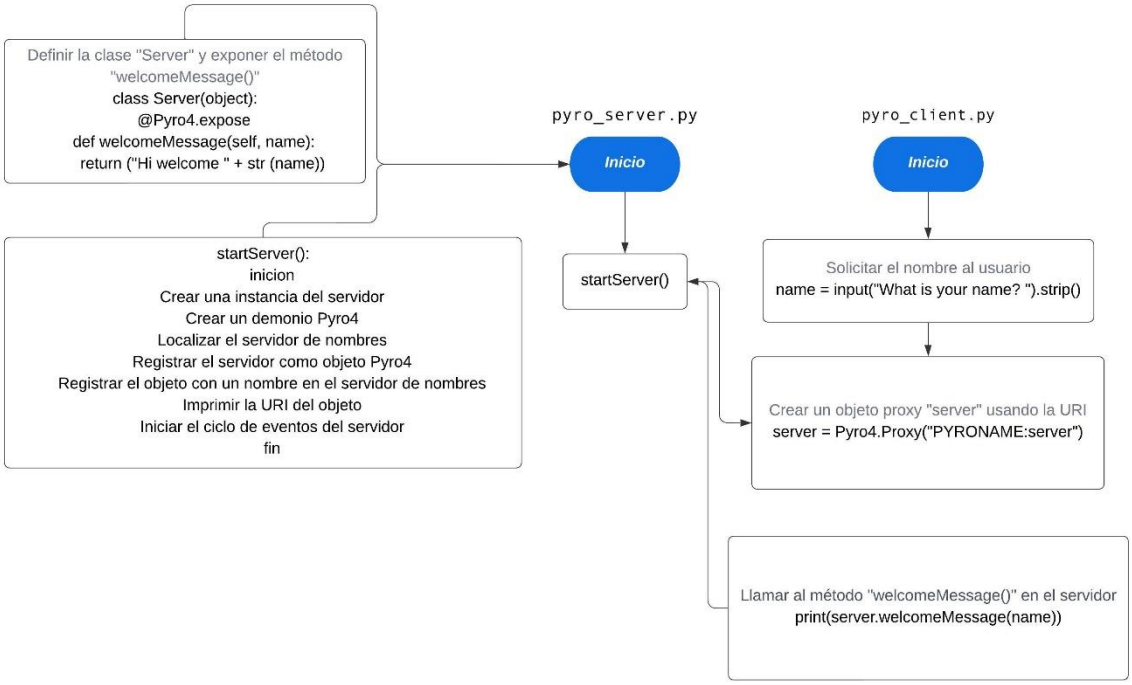
- El cliente comienza creando un objeto de socket utilizando la función socket(). Luego, obtiene el nombre de la máquina local (host) y establece el número de puerto en 60000.
- El cliente se conecta al servidor en el puerto 60000 utilizando s.connect((host, port)).
- El cliente envía el mensaje 'HelloServer!' al servidor codificado en bytes utilizando s.send('HelloServer!'.encode()).
- Luego, el cliente abre un archivo llamado 'received.txt' en modo binario ('wb') utilizando open(). Este archivo se utilizará para recibir los datos del servidor.
- El cliente entra en un bucle infinito con while True: para recibir continuamente datos del servidor.
- El cliente recibe datos del servidor utilizando s.recv(1024). Esto permite que el cliente reciba hasta 1024 bytes de datos.
- Los datos recibidos se imprimen en la consola.
- Luego, el cliente escribe los datos recibidos en el archivo 'received.txt' utilizando f.write(data).
- El cliente sigue recibiendo datos hasta que ya no haya más datos para recibir.

- Una vez que se han recibido todos los datos del servidor, el cliente cierra el archivo utilizando f.close().
- El cliente imprime un mensaje indicando que ha recibido con éxito el archivo.
- Finalmente, el cliente cierra la conexión con el servidor utilizando s.close().



- Importa el módulo Celery, que es necesario para trabajar con tareas distribuidas.
- Crea una instancia de Celery con el nombre "addTask" y especifica el broker como "amqp://guest@localhost//". El broker es el mecanismo que se utiliza para enviar y recibir mensajes entre el cliente y el servidor de Celery.
- Define una tarea llamada "add" que toma dos argumentos, "x" e "y", y devuelve su suma.
- Importa la tarea "add" desde el propio archivo. Esto permite ejecutar la tarea de suma directamente desde este script.
- se llama a la tarea "add" utilizando el método "delay" con los argumentos 5 y 5. Esto envía una solicitud para ejecutar la tarea "add" con los valores 5 y 5 como argumentos.

La ejecución de la tarea "add" se realiza de manera asíncrona a través de Celery, por lo que no bloquea la ejecución del programa principal.



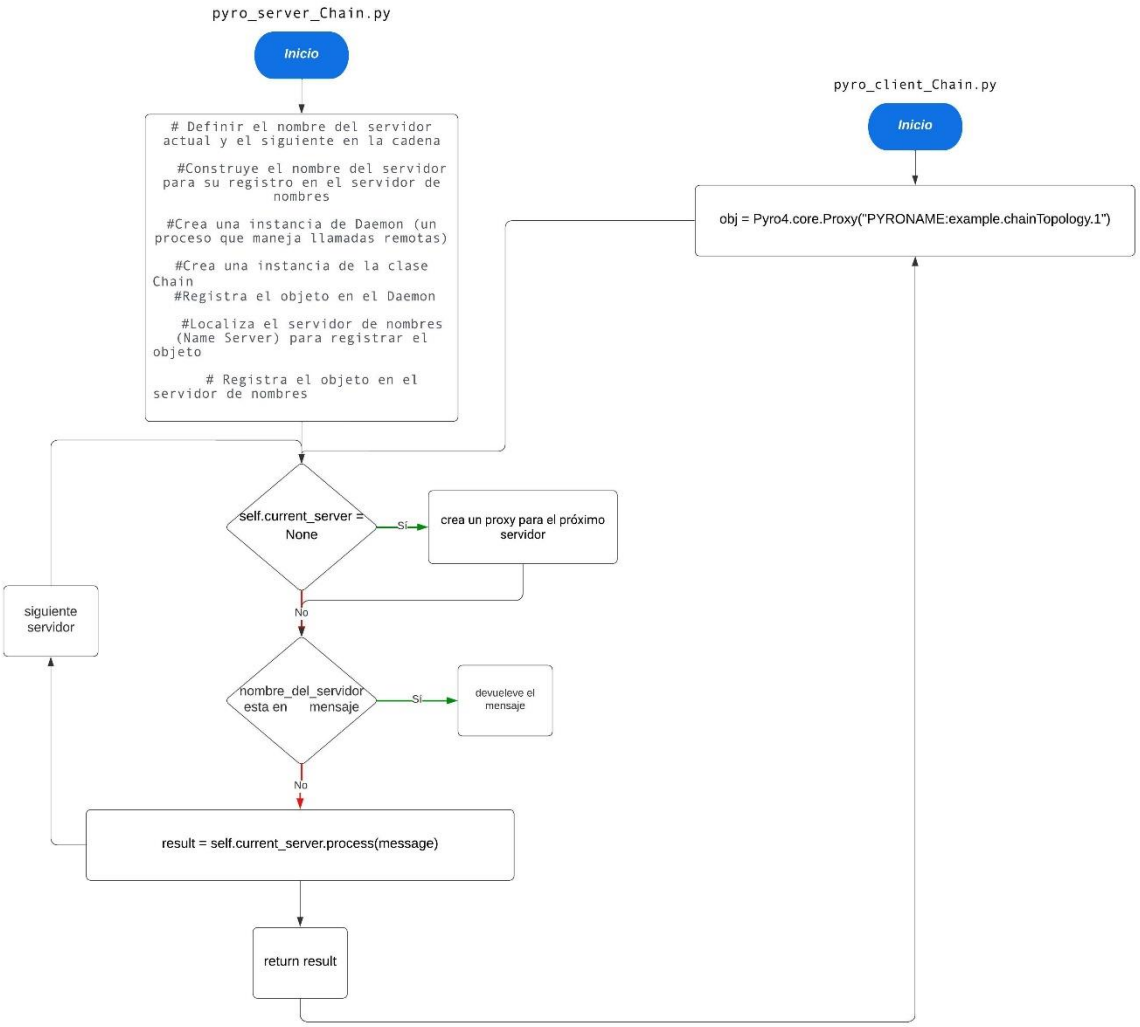
pyro_server.py

- Importa el módulo Pyro4 que permite la comunicación y la exposición de objetos remotos.
- Define una clase llamada "Server" que contiene un método llamado "welcomeMessage" decorado con @Pyro4.expose. Esto significa que el método se puede llamar de forma remota.
- Define una función llamada "startServer" que realiza las siguientes acciones:
    - Crea una instancia de la clase "Server".
    - Crea un objeto de demonio Pyro4 que gestionará las llamadas remotas.
    - Localiza un servidor de nombres (name server) que permitirá registrar objetos y buscarlos por nombre. d. Registra el objeto "server" como un objeto Pyro4 para que pueda ser accesible de forma remota.
    - Registra el objeto en el servidor de nombres con el nombre "server".
    - Imprime la URI (Uniform Resource Identifier) del objeto para que el cliente pueda utilizarlo.
    - Inicia el ciclo de eventos del demonio para esperar llamadas remotas.

pyro_client.py

- Importa el módulo Pyro4.
- Solicita al usuario que ingrese su nombre.
- Crea un objeto proxy llamado "server" utilizando la URI que debería proporcionar el servidor o el servidor de nombres. En este caso, la URI se construye con "PYRONAME:server", lo que significa que se buscará el objeto registrado con el nombre "server".

- Llama al método "welcomeMessage" en el objeto "server", pasando el nombre del usuario como argumento.
- Imprime el mensaje de bienvenida devuelto por el servidor.



Código de la clase Chain (chainTopology.py):

- Se importa la biblioteca Pyro4 y se usa el decorador @Pyro4.expose para marcar la clase Chain como expuesta, lo que permite que sus métodos sean invocados remotamente.
- El constructor __init__ de la clase Chain recibe el nombre y el servidor actual como parámetros y los asigna a atributos.
- El método process se encarga de procesar un mensaje. Si el servidor actual aún no se ha inicializado, se crea un proxy para el próximo servidor en la cadena utilizando el nombre registrado en Pyro4.
- Si el nombre del servidor actual está presente en el mensaje, se imprime un mensaje indicando que la cadena está cerrada y se devuelve una lista con un mensaje de completado en el servidor actual.

- De lo contrario, se imprime un mensaje indicando que el servidor actual reenvía el mensaje al próximo servidor en la cadena, se agrega el nombre del servidor actual al mensaje y se llama al método process del próximo servidor. El resultado se modifica para incluir el nombre del servidor actual y se devuelve.

Código del cliente (client_chain.py):

- Se importa la biblioteca Pyro4.
- Se crea un proxy para el objeto example.chainTopology.1 (el primer servidor en la cadena).
- Se realiza una llamada al método process del objeto proxy con un mensaje inicial "hello" y se imprime el resultado.

Código de los servidores (server_chain_X.py, donde X es 1, 2 o 3):

- Se importa la biblioteca Pyro4 y el módulo chainTopology que contiene la clase Chain.
- Se definen las variables current_server y next_server para indicar el servidor actual y el próximo servidor en la cadena.
- Se construye el nombre del servidor actual utilizando example.chainTopology. más el número del servidor.
- Se crea un daemon Pyro4 para el servidor.
- Se instancia un objeto de la clase Chain con los nombres de servidor actual y próximo servidor como parámetros.
- Se registra el objeto en el daemon y se registra su nombre en el Name Server (NS).
- Se imprime un mensaje indicando que el servidor se ha iniciado y se entra en el bucle de servicio del daemon.