TALLER #4 DE Sistemas Distribuidos-Comunicación

entre procesos.

COMUNICACIÓN ENTRE PROCESOS

Documentación del ejercicio y explicación detallada.

Presentado por: Havid Alvarez

Presentado al profesor: ANDRES ERNESTO DIAZ ORTEGA

Informe técnico incluye:

- Explicación de cada método implementado
- Comparación de rendimiento y casos de uso
- Capturas de pantalla de la ejecución
- Conclusiones sobre el uso de cada método

Desarrollo del informe

1.1 Implementación con Sockets

- Explicación del método:

Ya tenemos un servidor que acepta múltiples conexiones.

Ya tenemos clientes que envían y reciben mensajes.

La diferencia es que en vez de enviar "mensajes de chat", ahora el servidor debe mandar “tareas” y los clientes deben:

Recibir esa tarea, Procesarla, Enviar el resultado de vuelta al servidor.

Servidor:

server.py > ...

```python
import socket
import threading

HOST = "127.0.0.1"
PORT = 65432

# Aquí guardaremos los resultados de los clientes
resultados = {}  # {addr: resultado}


def manejar_cliente(conn, addr, tarea):
    """
    Función que se ejecuta en un hilo por cada cliente conectado.
    1. Envía la tarea al cliente.
    2. Recibe el resultado.
    3. Lo guarda en el diccionario resultados.
    """
    print(f"[NUEVA CONEXIÓN] {addr} conectado.")
    try:
        # 1. Enviar la tarea al cliente
        conn.sendall(tarea.encode("utf-8"))

        # 2. Esperar el resultado del cliente
        resultado = conn.recv(1024).decode("utf-8")
        resultados[addr] = resultado
        print(f"[RESULTADO] {addr}: {resultado}")
    except:
        print(f"[ERROR] con {addr}")
    finally:
        conn.close()

```

```python
def iniciar_servidor():
    """
    Inicia el servidor en localhost:65432
    y asigna la misma tarea a todos los clientes que se conecten.
    """
    servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    servidor.bind((HOST, PORT))
    servidor.listen()
    print(f"[ESCUCHANDO] Servidor en {HOST}:{PORT}")

    # Ejemplo: tarea que recibirán los clientes
    tarea = "2 + 2"

    while True:
        conn, addr = servidor.accept()
        hilo = threading.Thread(target=manejar_cliente, args=(conn, addr, tarea))
        hilo.start()


if __name__ == "__main__":
    iniciar_servidor()

```

Cliente:

client.py > ...

```python
import socket

HOST = "127.0.0.1"
PORT = 65432

# 1. Conectarse al servidor
cliente = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
cliente.connect((HOST, PORT))

# 2. Recibir la tarea del servidor
tarea = cliente.recv(1024).decode("utf-8")
print(f"[TAREA RECIBIDA] {tarea}")

# 3. Procesar la tarea (ejemplo: resolver usando eval)
try:
    resultado = str(eval(tarea))  #  Ojo: eval solo para pruebas, no en producción según chat gpt
except Exception as e:
    resultado = f"Error: {e}"

# 4. Enviar el resultado al servidor
cliente.sendall(resultado.encode("utf-8"))

# 5. Cerrar la conexión
cliente.close()


```

- Comparación de rendimiento y casos de uso:

Diferencia con el chat actual

-En el chat, el cliente escribe mensajes manualmente.

-Aquí, el servidor envía automáticamente una tarea, y el cliente la procesa y responde.

- Capturas de pantalla de la ejecución

- Conclusiones sobre el uso de cada método

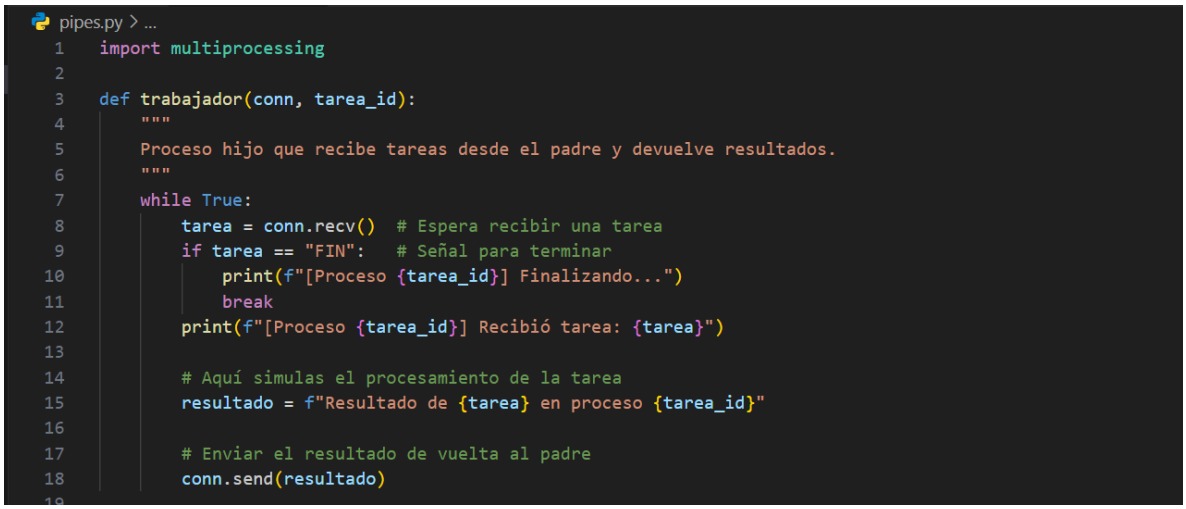
El uso de sockets TCP con threading permite implementar comunicación confiable entre un servidor y múltiples clientes. El servidor puede aceptar conexiones simultáneas, enviar tareas y recibir resultados gracias a la creación de hilos independientes para cada cliente. Este método es flexible y útil para aplicaciones como chats o sistemas distribuidos, aunque presenta limitaciones en escalabilidad y seguridad que deben considerarse en entornos más complejos.

1.2 Implementación con Pipes y Multiprocessing.

- Explicación del método:

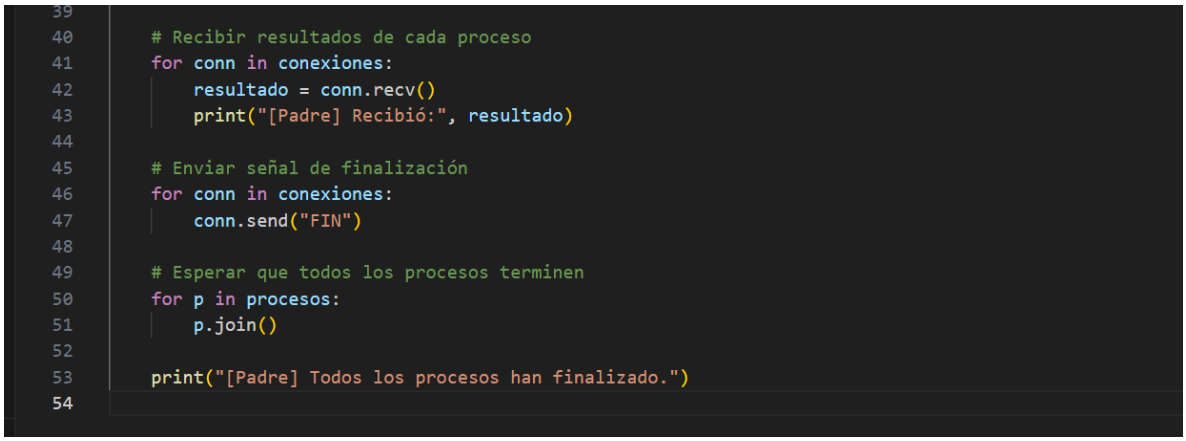
Este enfoque no usa sockets ni red, sino que trabaja dentro de la misma máquina, creando procesos hijos que se comunican con el padre mediante pipes.

Se crean procesos hijos con (multiprocessing.Process), cada hijo recibe tareas por un **Pipe** y devuelve resultados por el mismo Pipe, el padre reparte tareas, recibe resultados y finalmente envía "FIN" para cerrar.



```python
import multiprocessing

def trabajador(conn, tarea_id):
    """
    Proceso hijo que recibe tareas desde el padre y devuelve resultados.
    """
    while True:
        tarea = conn.recv()  # Espera recibir una tarea
        if tarea == "FIN":   # Señal para terminar
            print(f"[Proceso {tarea_id}] Finalizando...")
            break
        print(f"[Proceso {tarea_id}] Recibió tarea: {tarea}")

        # Aquí simulas el procesamiento de la tarea
        resultado = f"Resultado de {tarea} en proceso {tarea_id}"

        # Enviar el resultado de vuelta al padre
        conn.send(resultado)
```

```python


if __name__ == "__main__":
    procesos = []
    conexiones = []
    num_procesos = 3  # Número de procesos hijos

    # Crear procesos hijos con pipes
    for i in range(num_procesos):
        padre_conn, hijo_conn = multiprocessing.Pipe()
        p = multiprocessing.Process(target=trabajador, args=(hijo_conn, i))
        procesos.append(p)
        conexiones.append(padre_conn)
        p.start()

    # Enviar tareas a cada proceso
    for i, conn in enumerate(conexiones):
        tarea = f"Tarea {i+1}"
        print(f"[Padre] Enviando: {tarea}")
        conn.send(tarea)
```



```
39
40      # Recibir resultados de cada proceso
41      for conn in conexiones:
42          resultado = conn.recv()
43          print("[Padre] Recibió:", resultado)
44
45      # Enviar señal de finalización
46      for conn in conexiones:
47          conn.send("FIN")
48
49      # Esperar que todos los procesos terminen
50      for p in procesos:
51          p.join()
52
53      print("[Padre] Todos los procesos han finalizado.")
54
```

- Comparación de rendimiento y casos de uso:

Rendimiento:

Comunicación más rápida (memoria compartida en el mismo sistema operativo).

Se limita a procesos locales en una sola máquina.

Ideal cuando quieres paralelismo en CPU (ejecutar procesos en paralelo dentro de tu equipo).

Casos de uso:

Procesamiento intensivo de datos en paralelo.

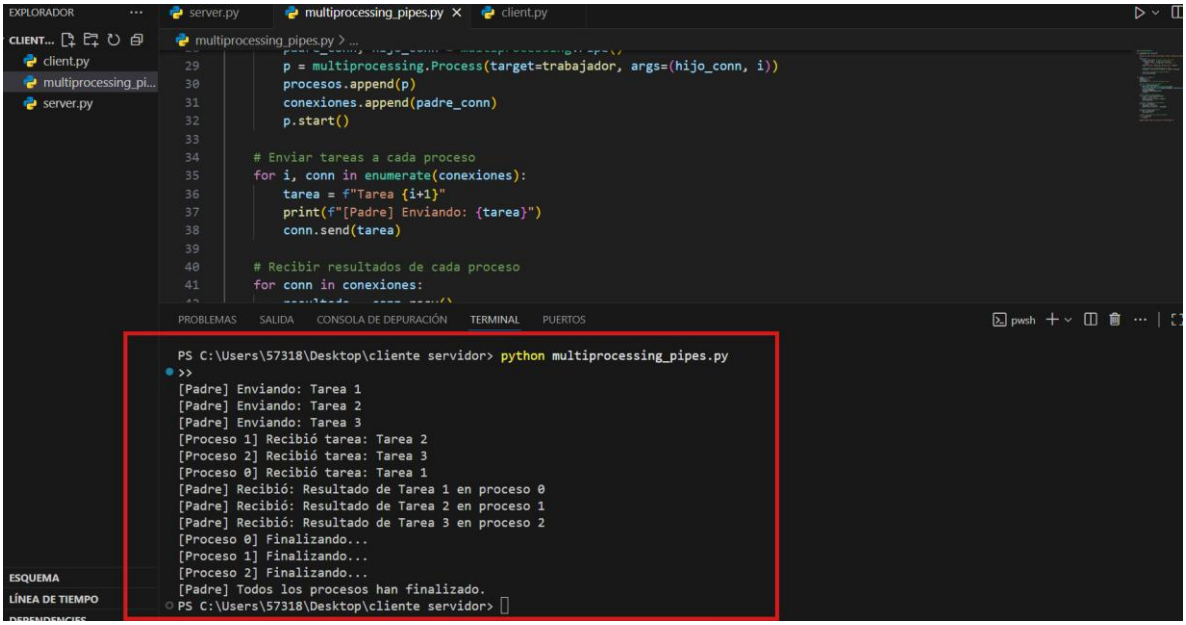
Algoritmos que pueden dividirse en subtareas (ej. cálculos científicos, análisis de datos).

Simulaciones o procesos batch en un solo equipo.

Comunicación eficiente entre procesos locales.

- Capturas de pantalla de la ejecución:

El padre reparte tareas, los hijos responden con resultados y luego se cierran correctamente.



- Conclusiones sobre el uso de cada método

La implementación con Pipes y Multiprocessing permite aprovechar el paralelismo de la CPU mediante procesos que se comunican de forma rápida y eficiente en un mismo equipo. Es ideal para distribuir tareas locales y consolidar resultados sin necesidad de una red, aunque no es aplicable a entornos distribuidos.

1.3 Implementación con Colas (Queues)

- Explicación del método:

multiprocessing.Queue()

Es una estructura segura para que varios procesos se comuniquen y compartan datos sin necesidad de sincronización manual.

Cola de tareas (cola_tareas)

El proceso padre coloca aquí los trabajos que deben ejecutarse.

Los procesos trabajadores (workers) los toman uno por uno para procesarlos.

Cola de resultados (cola_resultados)

Una vez que un worker termina su tarea, coloca el resultado en esta cola.

El proceso padre recoge los resultados desde aquí.

Workers (procesos hijos)

Cada worker corre en paralelo como un proceso independiente.

Su función es tomar tareas de la cola, procesarlas y enviar de vuelta los resultados.

Proceso padre

Envia las tareas a la cola de trabajo.

Recolecta los resultados de la cola de resultados.

Cuando ya no hay más trabajo, envía un mensaje especial ("FIN") para que cada worker se detenga correctamente.

las colas permiten un sistema organizado de productores (padre) y consumidores (workers), evitando bloqueos y haciendo el código más escalable.

Este método es más ordenado y escalable que Pipes, porque puedes tener múltiples productores y consumidores sin preocuparte tanto por la sincronización.

Queues.py > ...

```python
import multiprocessing
import time
import random

# --- Worker (proceso hijo) ---
def worker(cola_tareas, cola_resultados):
    """
    Función que ejecuta cada worker:
    - Toma tareas de la cola de tareas
    - Procesa la tarea
    - Devuelve el resultado en la cola de resultados
    """
    for tarea in iter(cola_tareas.get, "FIN"):  # Sale cuando recibe "FIN"
        print(f"[Worker {multiprocessing.current_process().name}] Procesando tarea: {tarea}")
        time.sleep(random.uniform(0.5, 1.5))  # Simula trabajo
        resultado = f"Tarea {tarea} completada"
        cola_resultados.put(resultado)

    print(f"[Worker {multiprocessing.current_process().name}] Finalizando...")

# --- Proceso principal (padre) ---
if __name__ == "__main__":
    num_workers = 3

    # Colas compartidas
    cola_tareas = multiprocessing.Queue()
    cola_resultados = multiprocessing.Queue()

    # Crear y arrancar procesos workers
    workers = []
    for i in range(num_workers):
        p = multiprocessing.Process(target=worker, args=(cola_tareas, cola_resultados))
        p.start()
        workers.append(p)

    # Enviar tareas a la cola
    tareas = [f"T{i}" for i in range(1, 11)]  # 10 tareas
    for t in tareas:
        cola_tareas.put(t)

    # Enviar señal de finalización ("FIN") a cada worker
    for _ in range(num_workers):
        cola_tareas.put("FIN")

    # Recoger resultados
    for _ in tareas:
        resultado = cola_resultados.get()
        print(f"[Padre] Resultado recibido: {resultado}")

    # Esperar a que todos los workers terminen
    for p in workers:
        p.join()

    print("[Padre] Todas las tareas han finalizado.")
```

Se crean las colas

cola_tareas: donde el padre coloca los trabajos.

cola_resultados: donde los workers dejan los resultados.

Se lanzan los workers

Cada worker se queda escuchando la cola hasta que recibe "FIN".

El padre envía tareas

Coloca varias tareas en la cola de trabajo.

Los workers procesan

Cada worker toma una tarea, la procesa y coloca el resultado en la cola de resultados.

El padre recoge resultados

Saca cada resultado de la cola y los imprime.

Finalización ordenada

Se envía "FIN" a cada worker para detenerlos correctamente.

- Comparación de rendimiento y casos de uso

Queues : Un poco más lentas que pipes, pero seguras y fáciles de usar (internamente usan locks).

Caso de uso:

Ideal cuando hay muchos procesos/trabajadores, distribución de tareas y recolección de resultados.

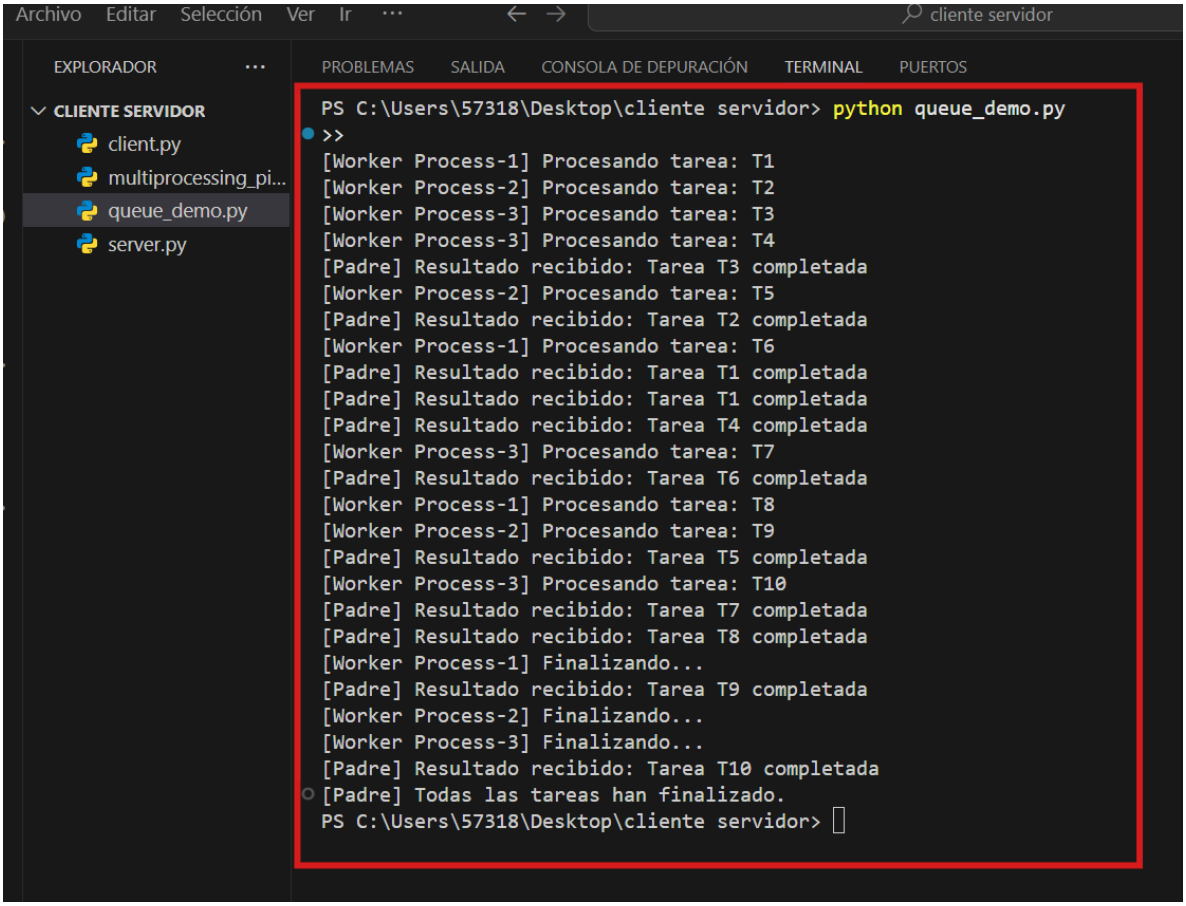
- Capturas de pantalla de la ejecución

Resultado de ejecución en terminal

Los workers procesando las tareas (números).

El padre recibiendo los resultados al cuadrado.

Mensaje final cuando todos los procesos terminan.



- Conclusiones sobre el uso de cada método

Sencillez: Queue facilita la comunicación entre procesos sin necesidad de manejar sockets o pipes manualmente.

Sincronización automática: gestiona de forma segura el acceso concurrente (evita bloqueos y corrupción de datos).

Escalabilidad: ideal cuando varios procesos producen/consumen tareas.

Limitación: solo funciona entre procesos locales (en la misma máquina), no entre máquinas distintas (para eso → sockets).

Comparación: Sockets vs Pipes vs Queues

Rendimiento

Pipes: Más rápidos, comunicación ligera entre pocos procesos.

Queues: Un poco más lentas que pipes, pero seguras y fáciles de usar (internamente usan locks).

Sockets: Más lentos (tienen sobrecarga de red), pero permiten comunicación local y remota.

Casos de uso

Pipes: Comunicación simple entre procesos padre ↔ hijo.

Queues: Ideal cuando hay muchos procesos/trabajadores, distribución de tareas y recolección de resultados.

Sockets: Cuando necesitas conexión entre máquinas o entre programas distintos (no solo procesos de Python).

- Resumen final de los métodos utilizados

Usa Pipes para cosas muy básicas y rápidas.

Usa Queues para multiprocessing dentro de una misma máquina.

Usa Sockets si necesitas red o clientes externos.