Sistema de Pedidos de Cafetería usando Computación Paralela y Distribuida en Python

Paula Andrea Caballero Sepúlveda
Mariana Ruge Vargas
Universidad Sergio Arboleda

***Resumen*—Este proyecto presenta la implementación de un sistema de pedidos de cafetería utilizando programación en Python. Se emplean sockets y hilos para construir una arquitectura distribuida y concurrente compuesta por tres entidades principales: cliente, caja y cocina. Se busca simular un flujo realista de pedidos, procesamiento y notificación, demostrando el uso de la computación paralela y distribuida para mejorar la eficiencia del sistema.**

I. Introducción

La computación paralela y distribuida permite diseñar sistemas que pueden atender múltiples procesos simultáneamente y garantizar la eficiencia de los recursos durante la ejecución de la misma, lo cual es esencial en contextos de atención al cliente constante y masiva como una cafetería. En este proyecto se desarrolló un sistema cliente-servidor basado en sockets TCP (Protocolo de transmisión de información) en Python, donde múltiples clientes realizan pedidos, los cuales son gestionados por una caja (intermediaria) puesto que registra y cobra los pedidos y luego procesados por una cocina (servidor de backend), luego regresa al cliente por medio de una solicitud final.

El presente informe muestra los resultados de esta simulación por medio de Sockets (puntos finales de comunicación por medio de la red) intecambiando datos de forma bidireccional.

II. Objetivos

II-A. Objetivo General

Desarrollar un sistema de pedidos de cafetería que utilice computación paralela y distribuida para simular un entorno de atención concurrente con múltiples clientes.

II-B. Objetivos Específicos

- Implementar comunicación cliente-servidor usando sockets TCP en Python.
- Utilizar hilos para permitir la atención simultánea de múltiples clientes.
- Simular la preparación de pedidos y la notificación asincrónica a cada cliente.
- Demostrar la implementación de algoritmos paralelos y distribuidos en contextos reales.

III. Metodología

Se siguió una metodología incremental, comenzando con la creación de tres componentes principales:

- **Cliente**: genera un pedido y espera una notificación.
- **Caja**: recibe pedidos de múltiples clientes y los reenvía a la cocina.
- **Cocina**: procesa los pedidos y notifica directamente a cada cliente.

Cada entidad corre en un hilo o proceso independiente y se comunica mediante sockets TCP. Se implementó un protocolo de mensajes con el formato: `pedido;cliente_id;host;puerto_respuesta`.

III-A. Desarrollo

A continuación se presenta el flujo del socket para la implementación de la computación Paralela y Distribuida.

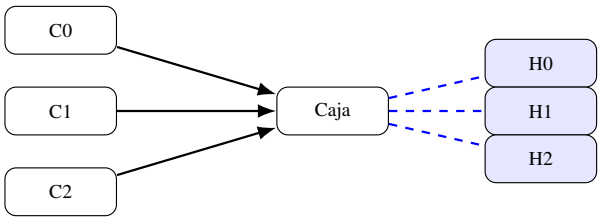


Figura 1: Flujo de procesamiento paralelo y distribuido por parte del sistema de la cafetería

Donde C reprsentan los **clientes** del sistema que pueden realizar pedidos simultaneamente. Estos pasan por la caja, y H simbolizan los **hilos** a los que fueron asignados para atender a los clientes de forma independiente. Esto permite que los procesos se realicen en **paralelo** sin bloquearse.

IV. Marco Teórico

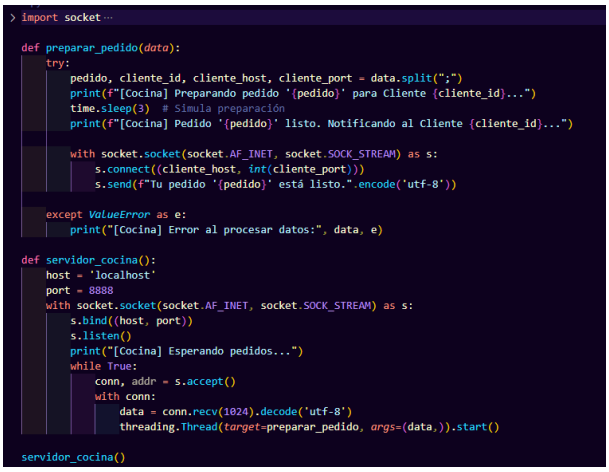
La **computación paralela** permite que múltiples tareas se ejecuten al mismo tiempo en un mismo sistema, aprovechando los recursos de hardware mediante el uso de hilos o procesos. En este proyecto, la paralelización se logra mediante el uso del módulo `threading` en Python, lo que permite atender varios clientes simultáneamente sin bloquear la ejecución principal de la caja.

```
while True:
    conn, addr = s.accept()
    threading.Thread(target=manejar_cliente, args=(conn,)).start()
```

Figura 2: Hilo en la implementación

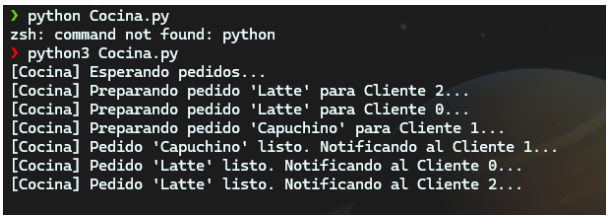
Por otro lado, la **computación distribuida** se refiere a sistemas donde diferentes procesos se ejecutan en distintas

máquinas o entornos, comunicándose entre sí a través de una red. En nuestro caso, los componentes cliente, caja y cocina simulan nodos distribuidos que intercambian mensajes a través de sockets, incluso cuando todos corren en la misma máquina local (`localhost`). En este caso, se usan dos terminales diferentes en las que cada una lleva un proceso diferente.



```
import socket ···

def preparar_pedido(data):
    try:
        pedido, cliente_id, cliente_host, cliente_port = data.split(";")
        print(f"[Cocina] Preparando pedido '{pedido}' para Cliente {cliente_id}...")
        time.sleep(3)  # Simula preparación
        print(f"[Cocina] Pedido '{pedido}' listo. Notificando al Cliente {cliente_id}...")

        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((cliente_host, int(cliente_port)))
            s.send(f"Tu pedido '{pedido}' está listo.".encode('utf-8'))

    except ValueError as e:
        print("[Cocina] Error al procesar datos:", data, e)

def servidor_cocina():
    host = 'localhost'
    port = 8888
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((host, port))
        s.listen()
        print("[Cocina] Esperando pedidos...")
        while True:
            conn, addr = s.accept()
            with conn:
                data = conn.recv(1024).decode('utf-8')
                threading.Thread(target=preparar_pedido, args=(data,)).start()

servidor_cocina()
```

Figura 3: Código implementación de la cocina

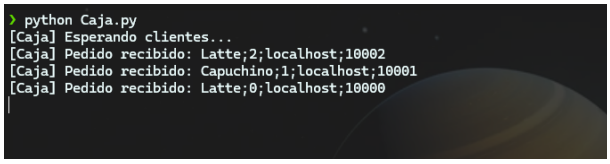


```
❯ python Cocina.py
zsh: command not found: python
❯ python3 Cocina.py
[Cocina] Esperando pedidos...
[Cocina] Preparando pedido 'Latte' para Cliente 2...
[Cocina] Preparando pedido 'Latte' para Cliente 0...
[Cocina] Preparando pedido 'Capuchino' para Cliente 1...
[Cocina] Pedido 'Capuchino' listo. Notificando al Cliente 1...
[Cocina] Pedido 'Latte' listo. Notificando al Cliente 0...
[Cocina] Pedido 'Latte' listo. Notificando al Cliente 2...
```

Figura 4: Ejecución de la cocina

```
import socket
import threading

def manejar_cliente(conn):
    data = conn.recv(1024).decode('utf-8')
    print(f"[Caja] Pedido recibido: {data}")
    conn.send("Tu pedido fue enviado a la cocina. Espera notificación.".encode('utf-8'))

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as cocina:
        cocina.connect(('localhost', 8888))  # Conectar con la Cocina
        cocina.send(data.encode('utf-8'))  # Reenviar datos completos

def servidor_caja():
    host = 'localhost'
    port = 9999
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((host, port))
        s.listen()
        print("[Caja] Esperando clientes...")
        while True:
            conn, addr = s.accept()
            threading.Thread(target=manejar_cliente, args=(conn,)).start()

servidor_caja()
```

Figura 5: Código implementación de la caja

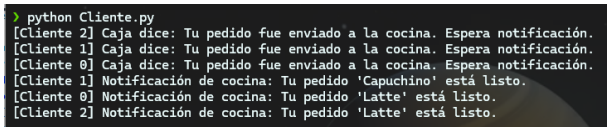


```
❯ python Caja.py
[Caja] Esperando clientes...
[Caja] Pedido recibido: Latte;2;localhost;10002
[Caja] Pedido recibido: Capuchino;1;localhost;10001
[Caja] Pedido recibido: Latte;0;localhost;10000
```

Figura 6: Ejecución de la caja

```
import socket
import threading
import random
import time

def escuchar_cliente(cliente_id, puerto_cliente):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind(("localhost", puerto_cliente))
        s.listen(1)
        conn, addr = s.accept()
        with conn:
            mensaje = conn.recv(1024).decode('utf-8')
            print(f"[Cliente {cliente_id}] Notificación de cocina: {mensaje}")

def cliente(cliente_id):
    pedidos = ["Capuchino", "Latte", "Americano"]
    pedido = random.choice(pedidos)
    puerto_cliente = 10000 + cliente_id  # Cada cliente escucha en un puerto diferente

    threading.Thread(target=escuchar_cliente, args=(cliente_id, puerto_cliente)).start()
    time.sleep(0.5)  # Esperar a que el servidor cliente esté escuchando

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect(('localhost', 9999))  # Conectarse a Caja
        mensaje = f"{pedido};{cliente_id};localhost;{puerto_cliente}"
        s.send(mensaje.encode('utf-8'))
        respuesta = s.recv(1024).decode('utf-8')
        print(f"[Cliente {cliente_id}] Caja dice: {respuesta}")

# Crear varios clientes
for i in range(3):
    threading.Thread(target=cliente, args=(i,)).start()
```

Figura 7: Código de la implementación del cliente



```
❯ python Cliente.py
[Cliente 2] Caja dice: Tu pedido fue enviado a la cocina. Espera notificación.
[Cliente 1] Caja dice: Tu pedido fue enviado a la cocina. Espera notificación.
[Cliente 0] Caja dice: Tu pedido fue enviado a la cocina. Espera notificación.
[Cliente 1] Notificación de cocina: Tu pedido 'Capuchino' está listo.
[Cliente 0] Notificación de cocina: Tu pedido 'Latte' está listo.
[Cliente 2] Notificación de cocina: Tu pedido 'Latte' está listo.
```

Figura 8: Ejecución del cliente

V. Resultados

Durante la ejecución del sistema se obtuvieron los siguientes resultados:

- Los clientes pudieron conectarse de forma concurrente al servidor Caja sin interferencias.
- La Caja creó un hilo para cada cliente, lo que permitió el reenvío inmediato de los pedidos a la Cocina sin retrasar nuevas conexiones.
- La Cocina procesó los pedidos en hilos independientes, enviando notificaciones personalizadas a cada cliente.

Podemos ver la gestión de forma paralela en una simplificación de este gráfico, que representa el comportamiento de los hilos. Como podemos ver, la mayoría de los hilos pueden ser atendidos en tiempo simultaneo, asignando diferentes recursos a su ejecución.

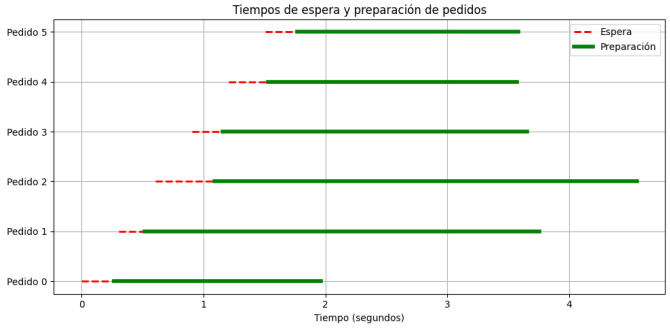


Figura 9: Ejecución de los hilos

Para la parte de la computación distribuida, podemos ver el tiempo de comunicación entre la caja, el cliente, y la cocina por medio de la siguiente gráfica.

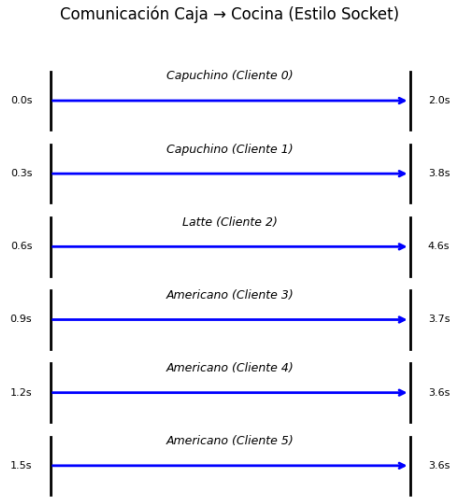


Figura 10: Comunicación

Se puede ver el tiempo de espera y respuesta entre cada uno varía dependiendo de la disponibilidad de los recursos mientras se hace la ejecución y recepción del pedido por medio de la caja.

VI. Análisis de Resultados

Los datos observados durante la ejecución muestran que los hilos (‘threads‘) permiten que cada entidad (Cliente, Caja, Cocina) pueda ejecutar múltiples tareas concurrentemente:

- En la **Caja**, cada vez que un cliente se conecta, se crea un nuevo hilo para atender su pedido. Esto significa que mientras un cliente es atendido, otros pueden conectarse y ser procesados al mismo tiempo.
- En la **Cocina**, cada pedido recibido es gestionado por un hilo separado, lo cual permite simular la preparación de varios pedidos a la vez sin esperar a que uno termine para empezar el siguiente.
- En los **Clientes**, cada uno escucha en su propio hilo para recibir la notificación, sin bloquear la ejecución del resto del código cliente.

Esto refleja exactamente el modelo de computación paralela, donde múltiples hilos comparten recursos del mismo sistema para ejecutar tareas simultáneamente. Además, al distribuir la lógica entre componentes distintos (Cliente, Caja, Cocina) conectados por red, el sistema implementa también computación distribuida.

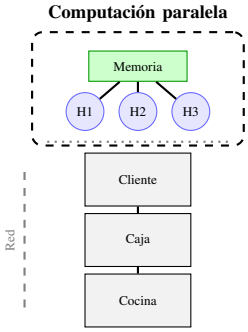


Figura 11: Modelo combinado de computación paralela y distribuida.

En resumen, los hilos permiten lograr una atención concurrente eficiente, mientras que la comunicación por sockets distribuye la carga entre diferentes nodos lógicos del sistema.

VII. Conclusiones

El proyecto demostró que la combinación de computación paralela (hilos) y distribuida (sockets) en Python permite construir sistemas eficientes y escalables. El uso de hilos fue fundamental para que cada componente pudiera atender múltiples eventos sin bloqueo. La arquitectura resultante es robusta, escalable y representa una base adecuada para sistemas reales de atención y notificación en línea. Se concluye que estos paradigmas son esenciales para construir aplicaciones modernas que gestionan múltiples usuarios simultáneamente.

Referencias

[1] Nestlé Professional. *Todo lo necesario para abrir una cafetería. Fase I: planeación*. 2 de diciembre de 2024. Disponible en: https://www.nestleprofessional-latam.com/co/tendencias-e-ideas/abrir-cafeteria. Consultado el 31 de mayo de 2025.

[2] Roser Vives Serra y Gonzalo Herrero Arroyo. *Operaciones y procesos en los servicios de bar y cafetería*. 1ª edición, 2014. ISBN: 978-84-907701-6-0. Disponible en: https://dmc2vm44yioo9.cloudfront.net/a17605fe-a581-4c59-9a1f-b925afd45fde.pdf.

[3] *threading — Paralelismo basado en hilos — documentación de Python - 3.8.20*. Disponible en: https://docs.python.org/es/3.8/library/threading.html. Consultado el 31 de mayo de 2025.

[4] Andrew S. Tanenbaum y Maarten Van Steen. *Distributed Systems: Principles and Paradigms*, 2ª edición, 2007. Disponible en: https://dl.acm.org/citation.cfm?id=1202502.

[5] Roman Trobec, Boštjan Slivnik, Patricio Bulić y Borut Robič. *Introduction to Parallel Computing*, 2018. DOI: 10.

1007/978-3-319-98833-7. Disponible en: https://doi.org/10.1007/978-3-319-98833-7.