

TCP_Server_Threadpool

Santiago Caicedo (2220035) y Daniel Convers (2221120)

February 17, 2026

1 Repositorio y material audiovisual

El código fuente completo del proyecto se encuentra disponible en un repositorio público de GitHub, donde se incluye la implementación del servidor TCP, la configuración utilizada para las pruebas y los archivos de persistencia generados durante la ejecución.

- Repositorio GitHub:
https://github.com/SantiagoMCAicedo08/Servidor_TCP
- Video demostrativo en YouTube:
<https://www.youtube.com/watch?v=4oMtdjchvMY>

En el video se muestra el funcionamiento del servidor, la ejecución de pruebas con Apache JMeter y la verificación de la persistencia de los datos generados.

2 Descripción del proyecto

En este proyecto se desarrolló un servidor TCP concurrente utilizando el lenguaje de programación Python. El objetivo principal fue implementar un servicio capaz de recibir múltiples solicitudes desde distintos clientes, procesarlas de manera concurrente mediante un *thread pool* y devolver una respuesta correcta en función de la lógica solicitada. Para la generación de carga y validación del comportamiento del servidor se utilizó la herramienta Apache JMeter.

3 Arquitectura del servidor

El servidor fue implementado utilizando el módulo `socket` de Python, lo que permite trabajar directamente con comunicación TCP de bajo nivel. El servidor se configura para escuchar en el puerto 5050 y aceptar conexiones desde cualquier interfaz de red, lo cual permite que los clientes se conecten tanto desde la misma máquina como desde equipos externos dentro de la red.

Para soportar múltiples clientes de manera simultánea, se utilizó la clase `ThreadPoolExecutor` del módulo `concurrent.futures`. Este enfoque permite

manejar la concurrencia de forma controlada, limitando el número máximo de hilos activos y evitando la creación descontrolada de threads. Cada conexión entrante es asignada a un hilo del pool, donde se ejecuta la lógica de procesamiento de la solicitud.

```
1 import socket
2 from concurrent.futures import ThreadPoolExecutor
3 from datetime import datetime
4 import threading
5
6 HOST = "0.0.0.0"
7 PORT = 5050
8 MAX_THREADS = 10
9 LOG_FILE = "server_log.txt"
10
11 log_lock = threading.Lock()
12
13 def is_prime(n):
14     if n < 2:
15         return False
16     for i in range(2, int(n ** 0.5) + 1):
17         if n % i == 0:
18             return False
19     return True
```

Listing 1: Servidor TCP con Thread Pool en Python

4 Procesamiento de solicitudes

Cada cliente se comunica con el servidor enviando una cadena de texto. El servidor recibe esta cadena, identifica el último carácter del mensaje y calcula cuántas veces dicho carácter aparece dentro de la cadena completa. Posteriormente, se evalúa si este número es un valor primo o no, utilizando una función de verificación optimizada que itera únicamente hasta la raíz cuadrada del número.

Con base en este análisis, el servidor genera una respuesta en el formato requerido, indicando el número de repeticiones del carácter y si dicho número es primo (*sí* o *no*). Esta respuesta es enviada de vuelta al cliente a través de la misma conexión TCP, y posteriormente la conexión es cerrada de forma controlada.

5 Manejo de concurrencia

El uso de un *thread pool* permite que múltiples clientes realicen solicitudes de forma paralela sin interferir entre sí. Cada conexión es independiente y se procesa en un hilo separado, garantizando que una solicitud lenta o bloqueada no afecte el desempeño general del servidor. Además, se establece un tiempo máximo de espera para la recepción de datos, lo que evita que el servidor quede bloqueado por clientes que no envían información.

```

1 def handle_client(client_socket, client_address):
2     print("CLIENT CONNECTED:", client_address) #          FIRST
3     print
4     try:
5         client_socket.settimeout(5)
6
7         data = client_socket.recv(1024)
8         if not data:
9             print("NO DATA RECEIVED") #          optional
10            return
11
12            message = data.decode("utf-8").strip()
13            print("RECEIVED:", message) #          SECOND print
14
15            last_char = message[-1]
16            count = message.count(last_char)
17            prime_result = "s " if is_prime(count) else "no"
18
19            response = f"{count} {prime_result}\n"
20            client_socket.sendall(response.encode("utf-8"))
21            print("SENT RESPONSE:", response.strip()) #          THIRD
22            print
23
24            client_socket.shutdown(socket.SHUT_WR)
25
26            timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
27            log_entry = f"{timestamp} | {client_address[0]} | {message} | {count} | {prime_result}\n"
28
29            with log_lock:
30                with open(LOG_FILE, "a", encoding="utf-8") as f:
31                    f.write(log_entry)
32
33            except socket.timeout:
34                print(f"TIMEOUT FROM {client_address}")
35
36            finally:
37                client_socket.close()
38                print("CONNECTION CLOSED") #          FOURTH print

```

Listing 2: ejecución de atención para cada cliente, cada una corriendo en su propio hilo

6 Persistencia de datos

Como parte del proyecto, se implementó un mecanismo de persistencia mediante el almacenamiento de cada solicitud procesada en un archivo de texto. Para cada conexión se registra la fecha y hora, la dirección IP del cliente, el mensaje recibido, el número de repeticiones del carácter analizado y el resultado de la verificación de primalidad.

Dado que el servidor opera en un entorno multihilo, se utilizó un mecanismo de sincronización mediante un Lock para asegurar que únicamente un hilo pueda

escribir en el archivo a la vez. Esto garantiza la integridad de los datos almacenados y evita condiciones de carrera durante el acceso concurrente al archivo de registro.

2026-02-16 10:37:12	192.168.101.123	abc2a	2	sí
2026-02-16 10:37:12	192.168.101.123	abc2a	2	sí
2026-02-16 10:37:12	192.168.101.123	abc10a	2	sí
2026-02-16 10:37:12	192.168.101.123	abc2a	2	sí
2026-02-16 10:37:12	192.168.101.123	abc10a	2	sí
2026-02-16 10:37:12	192.168.101.123	abc2a	2	sí
2026-02-16 10:37:12	192.168.101.123	abc10a	2	sí
2026-02-16 10:37:12	192.168.101.123	abc2a	2	sí
2026-02-16 10:37:12	192.168.101.123	abc10a	2	sí
2026-02-16 10:37:12	192.168.101.123	abc2a	2	sí
2026-02-16 10:37:12	192.168.101.123	abc10a	2	sí
2026-02-16 10:37:12	192.168.101.123	abc10a	2	sí
2026-02-16 22:54:43	127.0.0.1	abc6a	2	sí
2026-02-16 22:54:43	127.0.0.1	abc16a	2	sí
2026-02-16 22:54:43	127.0.0.1	abc1a	2	sí
2026-02-16 22:54:43	127.0.0.1	abc12a	2	sí
2026-02-16 22:54:43	127.0.0.1	abc5a	2	sí
2026-02-16 22:54:43	127.0.0.1	abc17a	2	sí
2026-02-16 22:54:43	127.0.0.1	abc2a	2	sí
2026-02-16 22:54:43	127.0.0.1	abc13a	2	sí

Figure 1: Manejo de persistencia (fecha y hora, IP del cliente, mensaje, cantidad de repeticiones, si es o no número primo)

7 Pruebas con Apache JMeter

Para la validación del funcionamiento del servidor se utilizó Apache JMeter, específicamente el componente *TCP Sampler*. Esta herramienta permitió simular múltiples clientes enviando solicitudes de manera concurrente, lo cual facilitó la evaluación del comportamiento del servidor bajo carga.

Mediante la configuración de grupos de hilos y contadores de iteraciones, fue posible generar múltiples solicitudes paralelas y observar tanto las respuestas del servidor como los tiempos de procesamiento. De esta manera se comprobó que el servidor responde correctamente a todas las solicitudes, maneja adecuadamente la concurrencia y mantiene la persistencia de los datos generados.

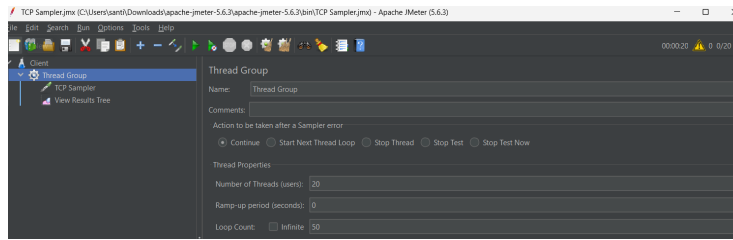


Figure 2: Configuración de simulación de multiples clientes Apache JMeter

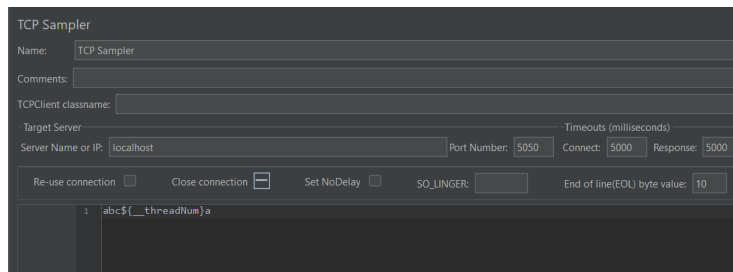


Figure 3: definición de parámetros de envío de mensajes y tiempos de espera Apache JMeter

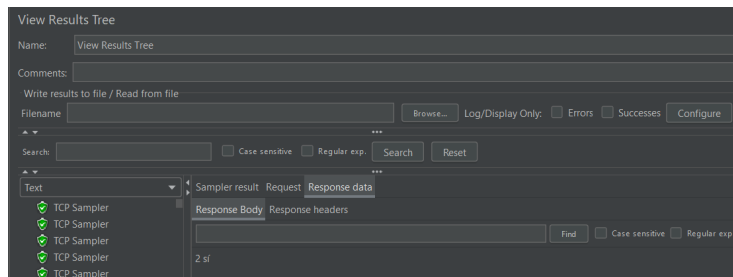


Figure 4: respuesta del servidor al cliente Apache JMeter

8 Conclusiones

El proyecto cumple con los objetivos propuestos, demostrando la correcta implementación de un servidor TCP concurrente con manejo eficiente de múltiples clientes, procesamiento adecuado de solicitudes, persistencia de datos y validación mediante herramientas de prueba profesionales. La arquitectura utilizada permite escalabilidad y un comportamiento estable incluso bajo múltiples conexiones simultáneas.