

Actividad 7:  
Ejercicios sobre comunicación entre procesos  
Cómputo de Alto Rendimiento  
**Luis Fernando Izquierdo Berdugo**

## Análisis del Código 1

Este código crea dos hilos concurrentes, donde cada uno ejecuta una carga de trabajo definida por un número “bignum” y muestra mensajes durante 10 iteraciones. La clase “MiHilo”:

- hereda de “Thread”,
- tiene la función “\_\_init\_\_” donde se inicializa la clase base, el número de operaciones a hacer y el nombre del hilo,
- tiene la función “run”, donde se hacen 10 iteraciones de impresión del nombre del Hilo y el número de iteración.

Al ejecutar la función “test”:

- Se definen la variables “num1” y “num2”
- Se crean los hilos “thr1” y “thr2” llamando a la clase “MiHilo” con su número de operaciones “num1” y “num2” respectivamente, así como sus nombres “Hilo 1” e “Hilo 2”.
- Se inician ambos hilos “.start()” (ejecutan su función “run”)
- Se espera a que terminen “.join()”

La salida será algo como en la imagen:

The screenshot shows a code editor with a Python file named `codigo1.py`. The code defines a `MiHilo` class that inherits from `Thread`. It has an `__init__` method that takes `num` and `nom` as arguments. The `run` method contains a nested loop: an outer loop for `l` in `range(10)` and an inner loop for `k` in `range(self.bignum)`. The inner loop calculates a sum `res` and prints the thread's name and iteration number. A `test` method creates two threads, `thr1` and `thr2`, with different `num` values.

```
1 import time
2 from threading import Thread
3 (class) MiHilo
4 class MiHilo(Thread):
5     def __init__(self, num, nom):
6         Thread.__init__(self)
7         self.bignum = num
8         self.nombre = nom
9
10    def run(self):
11        for l in range(10):
12            for k in range(self.bignum): # trabajo pesado
13                res = 0
14                for i in range(self.bignum):
15                    res += 1
16                print(self.nombre + ": iteracion " + str(l))
17
18    def test():
19        num1 = 1000
20        num2 = 500
21        thr1 = MiHilo(num1, "Hilo 1")
22        thr2 = MiHilo(num2, "Hilo 2")
23
```

The terminal output shows the execution of the program, displaying the iteration numbers for both threads. The output is as follows:

```
uis/Documents/Maestria/C mputo de Alto Rendimiento/Unidad 7/codigo1.py"
Hilo 2: iteracion 0
Hilo 2: iteracion 1
Hilo 2: iteracion 2
Hilo 2: iteracion 3
Hilo 2: iteracion 4
Hilo 1: iteracion 0
Hilo 2: iteracion 5
Hilo 2: iteracion 6
Hilo 2: iteracion 7
Hilo 2: iteracion 8
Hilo 2: iteracion 9
Hilo 1: iteracion 1
Hilo 1: iteracion 2
Hilo 1: iteracion 3
Hilo 1: iteracion 4
Hilo 1: iteracion 5
Hilo 1: iteracion 6
Hilo 1: iteracion 7
Hilo 1: iteracion 8
Hilo 1: iteracion 9
Hilo 1: iteracion 0
Hilo 2: iteracion 0
Hilo 2: iteracion 1
Hilo 2: iteracion 2
Hilo 2: iteracion 3
Hilo 2: iteracion 4
Hilo 2: iteracion 5
```

Si se ejecutara el c digo con ciclos "for", se tendr a una ejecuci n secuencial totalmente, lo cual aumentar a el tiempo total de ejecuci n. Se ejecutar a el primer conjunto de iteraciones, luego el segundo, etc. Teniendo una salida como:

```
Hilo 1 iteracion 0
Hilo 1 iteracion 1
Hilo 1 iteracion 2
Hilo 1 iteracion 3
Hilo 1 iteracion 4
Hilo 1 iteracion 5
Hilo 1 iteracion 6
Hilo 1 iteracion 7
Hilo 1 iteracion 8
Hilo 1 iteracion 9
Hilo 2 iteracion 0
Hilo 2 iteracion 1
Hilo 2 iteracion 2
Hilo 2 iteracion 3
Hilo 2 iteracion 4
Hilo 2 iteracion 5
```

```
Hilo 2 iteracion 6  
Hilo 2 iteracion 7  
Hilo 2 iteracion 8  
Hilo 2 iteracion 9
```

## Análisis del Código 2

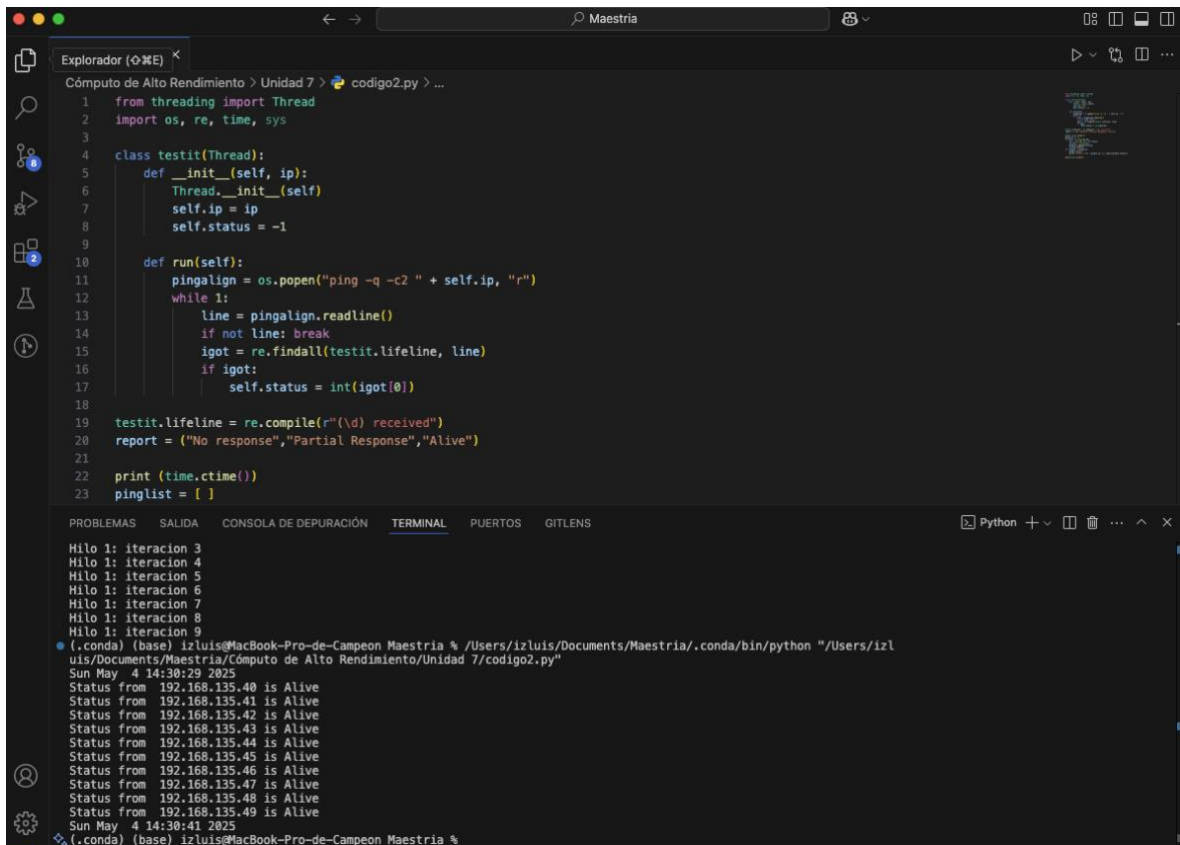
Este código crea una serie de hilos que se encargan de hacer ping a una dirección IP local distinta, al final se reporta si hubo una respuesta o no.

La clase “testit”:

- hereda de “Thread”,
- tiene la función “\_\_init\_\_” donde se guarda la dirección IP que se va a probar y pone el status en -1
- tiene la función “run”, donde se ejecuta el comando ping a la IP (con “os.popen”), lee la respuesta y usa una expresión regular ( `r"(\d) received"` ) para detectar cuantos paquetes se recibieron, finalmente guarda el resultado en “self.status” donde 0 es que no se recibió nada, 1 es que hubo respuesta parcial y 2 que hubo respuesta exitosa.

En la parte principal del programa se crea “current = testit(ip)” y lanzan (“current.start()”) 10 hilos (uno por cada IP), después espera a que cada hilo termine (“pingle.join()”) e imprime su resultado.

El resultado sería algo como en la imagen:

The image shows a VS Code editor window with a Python script named 'codigo2.py' and its terminal output. The script defines a 'testit' class that inherits from 'Thread'. It has an 'init' method that sets 'self.ip' and 'self.status'. The 'run' method uses 'os.popen' to execute a 'ping' command, reads the output, and updates 'self.status'. The terminal shows the output of running the script, displaying 'Hilo 1: iteracion' followed by 'Status from' and 'is Alive' for various IP addresses. The terminal also shows the command used to run the script: 'python "/Users/izluis/Documents/Maestria/Cómputo de Alto Rendimiento/Unidad 7/codigo2.py"'.

```
1 from threading import Thread
2 import os, re, time, sys
3
4 class testit(Thread):
5     def __init__(self, ip):
6         Thread.__init__(self)
7         self.ip = ip
8         self.status = -1
9
10    def run(self):
11        pingalign = os.popen("ping -q -c2 " + self.ip, "r")
12        while 1:
13            line = pingalign.readline()
14            if not line: break
15            igot = re.findall(testit.lifeline, line)
16            if igot:
17                self.status = int(igot[0])
18
19    testit.lifeline = re.compile(r"(\d) received")
20    report = ("No response", "Partial Response", "Alive")
21
22    print (time.ctime())
23    pinglist = [ ]
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS GITLENS

Hilo 1: iteracion 3  
Hilo 1: iteracion 4  
Hilo 1: iteracion 5  
Hilo 1: iteracion 6  
Hilo 1: iteracion 7  
Hilo 1: iteracion 8  
Hilo 1: iteracion 9

(.conda) (base) izluis@MacBook-Pro-de-Campeon Maestria % /Users/izluis/Documents/Maestria/.conda/bin/python "/Users/izluis/Documents/Maestria/Cómputo de Alto Rendimiento/Unidad 7/codigo2.py"  
Sun May 4 14:30:29 2025  
Status from 192.168.135.40 is Alive  
Status from 192.168.135.41 is Alive  
Status from 192.168.135.42 is Alive  
Status from 192.168.135.43 is Alive  
Status from 192.168.135.44 is Alive  
Status from 192.168.135.45 is Alive  
Status from 192.168.135.46 is Alive  
Status from 192.168.135.47 is Alive  
Status from 192.168.135.48 is Alive  
Status from 192.168.135.49 is Alive  
Sun May 4 14:30:41 2025  
(.conda) (base) izluis@MacBook-Pro-de-Campeon Maestria %

Si se ejecutara este código con ciclos for, el ping a cada IP se realizaría uno tras otro, esperando que termine cada uno de ellos, esto aumentaría el tiempo de espera. Al ejecutarlo con varios hilos, se tomó 12 segundos en total, si tomamos un tiempo de 2 segundos para cada proceso, tendríamos 20 segundos en total de manera secuencial y esto siendo positivos con el tiempo que tarda en hacerse el ping.

## Hilos literarios

El objetivo de este ejercicio es crear tres hilos que escriban el mismo archivo, sin embargo, debe ser de forma ordenada y sincronizada. Es necesario asegurar que:

1. Primero escriba Quijote
2. Romeo escriba de segundo
3. Julieta escriba de último

Es necesaria la sincronización porque, si los tres hilos escriben al mismo tiempo, el resultados podría tener frases mezcladas de cada escritor, ya que comparten el mismo recurso que es el archivo.

El código en python utiliza las librerías “threading” y “time” para ejecución.

```
from threading import Thread, Lock
import time
```

Lo primero será crear variables con los textos que escribirá cada escritor, con el fin de que el código general sea más simple de leer.

```
# Textos que escribirá cada hilo
quijote_texto = """...En un lugar de la Mancha de cuyo nombre no
quiero acordarme, no ha mucho tiempo que vivía
un hidalgo de los de lanza en astillero, adarga
antigua, rocín flaco y galgo corredor.
Una olla de algo más vaca que carnero, salpicón
las más noches, duelos y quebrantos los sábados,
lentejas los viernes...
"""

romeo_texto = """- Habla. ¡Oh! ¡Habla otra vez ángel resplan-
deciente!. . . Porque esta noche apareces tan
esplendorosa sobre mi cabeza como un ala-
do mensajero celeste ante los ojos estáticos y
maravillados de los mortales, que se inclinan
hacia atrás para verle, cuando él cabalga so-
bre las tardas perezosas nubes y navega en
el seno del aire.
"""

julietta_texto = """¡Oh Romeo, Romeo! ¿Por qué eres tú
Romeo? Niega a tu padre y rehusa tu nom-
bre; o, si no quieres, júrame tan sólo que
me amas, y dejaré yo de ser una Capuleto.
"""
```

Lo siguiente será crear las variables compartidas, que serán el lock para el acceso al archivo y el turno de cada uno, así como el archivo de salida.

```
# Variables compartidas
lock = Lock()
turno = 0 # 0 = Quijote, 1 = Romeo, 2 = Julieta
```

```
# Ruta del archivo a escribir
archivo_salida = "literatura.txt"
```

A continuación se creará la clase `Escritor`, esta hereda de `Thread` y en su inicialización se guardará su nombre, el texto y su turno

```
class Escritor(Thread):
    def __init__(self, nombre, texto, mi_turno):
        super().__init__()
        self.nombre = nombre
        self.texto = texto
        self.mi_turno = mi_turno
```

Dentro de la misma clase `Escritor`, se crea su método “run” que usa el turno global para compararlo con el del escritor, si no son iguales, espera 0.1 segundos para volver a revisar. Si son iguales, usa el lock para acceder al archivo y en este escribe el texto, finalmente le añade 1 a la variable global turno y pasar al siguiente escritor.

```
def run(self):
    global turno
    while True:
        if turno == self.mi_turno:
            with lock:
                with open(archivo_salida, "a", encoding="utf-8") as f:
                    f.write(f"\n[{self.nombre}]\n")
                    f.write(self.texto)
                turno += 1 # pasa al siguiente escritor
            break
        else:
            time.sleep(0.1) # Espera un poco antes de volver a revisar
```

Finalmente, en la parte principal del programa se crean los hilos de cada escritor, se lanzan y se espera a que terminen, después de esto, se imprime “Archivo escrito correctamente” en la consola.

```
# Crear hilos
quijote = Escritor("Quijote", quijote_texto, 0)
romeo = Escritor("Romeo", romeo_texto, 1)
```

```
julieta = Escritor("Julieta", julieta_texto, 2)

# Lanzar hilos
quijote.start()
romeo.start()
julieta.start()

# Esperar a que terminen
quijote.join()
romeo.join()
julieta.join()

print("Archivo escrito correctamente")
```

A continuación se ve la ejecución correcta del programa con el mensaje “Archivo escrito correctamente” en la consola.

```
30 # Variables compartidas
31 lock = Lock()
32 turno = 0 # 0 = Quijote, 1 = Romeo, 2 = Julieta
33
34 # Ruta del archivo a escribir
35 archivo_salida = "literatura.txt"
36
37 class Escritor(Thread):
38     def __init__(self, nombre, texto, mi_turno):
39         super().__init__()
40         self.nombre = nombre
41         self.texto = texto
42         self.mi_turno = mi_turno
43
44     def run(self):
45         global turno
46         while True:
47             if turno == self.mi_turno:
48                 with lock:
49                     with open(archivo_salida, "a", encoding="utf-8") as f:
50                         f.write(f"\n[{self.nombre}]\n")
51                         f.write(self.texto)
52                         turno += 1 # pasa al siguiente escritor
53                 break
54             else:
55                 time.sleep(0.1) # Espera un poco antes de volver a revisar
56
57 # Crear hilos
58 quijote = Escritor("Quijote", quijote_texto, 0)
59 romeo = Escritor("Romeo", romeo_texto, 1)
60 julieta = Escritor("Julieta", julieta_texto, 2)
61
62 quijote.start()
63 romeo.start()
64 julieta.start()
65
66 quijote.join()
67 romeo.join()
68 julieta.join()
69
70 print("Archivo escrito correctamente")
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS GITLENS

(base) izluis@MacBook-Pro-de-Campeon Maestria % /Users/izluis/Documents/Maestria/.conda/bin/python "/Users/izluis/Documents/Maestria/Cómputo de Alto Rendimiento/Unidad 7/codigo3.py"

Archivo escrito correctamente

(base) izluis@MacBook-Pro-de-Campeon Maestria %

Lín. 65, col. 16 Espacios: 4 UTF-8 LF Python 3.12.9 (conda) Go Live Prettier

Y al revisar el archivo generado, notamos que se generó correctamente:

```
codigo2.py U  codigo3.py U  literatura.txt U x
literatura.txt
1
2  [[Quijote]]
3  ...En un lugar de la Mancha de cuyo nombre no
4  quiero acordarme, no ha mucho tiempo que vivía
5  un hidalgo de los de lanza en astillero, adarga
6  antigua, rocín flaco y galgo corredor.
7  Una olla de algo más vaca que carnero, salpicón
8  las más noches, duelos y quebrantos los sábados,
9  lentejas los viernes...
10
11  [Romeo]
12  - Habla. ¡Oh! ¡Habla otra vez ángel resplan-
13  deciente!. . . Porque esta noche apareces tan
14  esplendorosa sobre mi cabeza como un ala-
15  do mensajero celeste ante los ojos estáticos y
16  maravillados de los mortales, que se inclinan
17  hacia atrás para verle, cuando él cabalga so-
18  bre las tardas perezosas nubes y navega en
19  el seno del aire.
20
21  [Julietta]
22  ¡Oh Romeo, Romeo! ¿Por qué eres tú
23  Romeo? Niega a tu padre y rehusa tu nom-
24  bre; o, si no quieres, júrame tan sólo que
25  me amas, y dejaré yo de ser una Capuleto.
26
```

## Cuenta Palabras

El objetivo es generar un programa que reciba varios archivos de texto como argumentos, cree un hilo por cada uno de ellos y cuente las palabras en paralelo. El output deberá contener el número de palabras por archivo, el total acumulado y debe contener el mismo orden en que se enviaron los archivos.

El código de Python usa las librerías `threading` y `sys`, esta última para abrir los archivos.

```
from threading import Thread
import sys
```

Primero se creará la clase “ContadorPalabras”, en su inicialización, se guarda el nombre del archivo a procesar, la lista donde se guardan los resultados y un índice de su posición en la lista anterior.

```
class ContadorPalabras(Thread):
    def __init__(self, archivo, resultados, indice):
        super().__init__()
```



```
self.archivo = archivo      # Nombre del archivo a procesar
self.resultados = resultados # Lista compartida para guardar los resultados
self.indice = indice        # Posición que le corresponde en esa lista
```

Lo siguiente es la función run, usa un try-except para ver si el archivo existe, en caso de que no exista imprime en la consola “Archivo no encontrado”. Si el archivo existe, lo abre en modo lectura, lee todo su contenido, guarda cada palabra en la variable “palabras” y la cantidad de palabras en la variable “cantidad”, finalmente guarda el resultado en la lista.

```
def run(self):
    try:
        # Abre el archivo en modo lectura
        with open(self.archivo, 'r', encoding='utf-8') as f:
            texto = f.read()      # Lee todo el contenido
            palabras = texto.split() # Separa por espacios (simplificado)
            cantidad = len(palabras) # Cuenta las palabras
            self.resultados[self.indice] = (self.archivo, cantidad) # Guarda el resultado
    except FileNotFoundError:
        # Si el archivo no existe, guarda un mensaje de error en su lugar
        self.resultados[self.indice] = (self.archivo, "Archivo no encontrado")
```

Dentro del main del programa, se valida que haya más de dos argumentos (para asegurarnos que mínimo haya un archivo a leer), en caso de que no haya, se imprimen instrucciones de uso del programa.

```
# Punto de entrada principal
if __name__ == "__main__":
    # Validación de argumentos
    if len(sys.argv) < 2:
        print("Uso: python cuenta_palabras.py archivo1 archivo2 ...")
        sys.exit(1)
```

Guarda variables con la lista de nombres que se pasaron en la ventana de comandos, una lista vacía con la misma cantidad de entradas y los hilos a utilizar

```
archivos = sys.argv[1:] # Lista de nombres de archivos desde la línea de comandos
resultados = [None] * len(archivos) # Lista vacía con la misma cantidad de entradas
hilos = []
```

Se crea un hilo por archivo y se inicializa, también se añade a la lista “hilos”.

```
for i, archivo in enumerate(archivos):
    hilo = ContadorPalabras(archivo, resultados, i)
    hilo.start()
    hilos.append(hilo)
```

Se espera a que todos los hilos terminen y se inicializa la variable “total\_palabras” en cero.

```
for hilo in hilos:
    hilo.join()

total_palabras = 0
```

Se imprimen los resultados y se calcula el total

```
for archivo, cantidad in resultados:
    print(f"{archivo}: {cantidad} palabras")
    if isinstance(cantidad, int): # Suma solo si no hubo error
        total_palabras += cantidad

print(f"total: {total_palabras} palabras")
```

A continuación se muestra la terminal con la correcta ejecución del archivo con distintos casos. El primer caso es cuando no tiene argumentos para archivos, el segundo caso cuando no encuentra los archivos, el tercer caso cuando solo encuentra un archivo y el último caso es el caso ideal.

```
❶ (.conda) (base) izluis@MacBook-Pro-de-Campeon Unidad 7 % python codigo4.py
Uso: python cuenta_palabras.py archivo1 archivo2 ...
❷ (.conda) (base) izluis@MacBook-Pro-de-Campeon Unidad 7 % python codigo4.py file1 file2 file3
file1: Archivo no encontrado palabras
file2: Archivo no encontrado palabras
file3: Archivo no encontrado palabras
total: 0 palabras
❸ (.conda) (base) izluis@MacBook-Pro-de-Campeon Unidad 7 % python codigo4.py file1.txt file2 file3
file1.txt: 12 palabras
file2: Archivo no encontrado palabras
file3: Archivo no encontrado palabras
total: 12 palabras
❹ (.conda) (base) izluis@MacBook-Pro-de-Campeon Unidad 7 % python codigo4.py file1.txt file2.txt file3.txt
file1.txt: 12 palabras
file2.txt: 9 palabras
file3.txt: 17 palabras
total: 38 palabras
```

## Cliente-Servidor a chat

El objetivo de este programa es modificar los archivos “cliente.py” y “server.py” para que actúen como un chat bidireccional y ya no como un cliente y un servidor.

Este es el código original del “server.py”

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 16 18:29:55 2015

@author: mag
"""

import socket
puerto = 4545;
miSocket = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
miSocket.bind( ( socket.gethostname(), puerto ) )
miSocket.listen( 1 )
while True:
    channel, details = miSocket.accept()
    channel.send( 'Hola Mundo!' )
    channel.close()
```

- Crea un socket TCP y lo vincula al puerto 4545
- Escucha una conexión
- Cuando un cliente se conecta, le envía el mensaje “Hola Mundo!” y cierra la conexión.

Este es el código original de “cliente.py”

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 16 18:28:33 2015

@author: mag
"""

import socket

miSocket = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
miSocket.connect( (socket.gethostname(), 4545 ) )
data, server = miSocket.recvfrom( 100 )
print data
```

```
miSocket.close()
```

- El cliente se conecta al mismo host y puerto que el server
- Espera hasta recibir 100 bytes
- Imprime el mensaje recibido
- Cierra la conexión

Primero se cambiará el código de server.py, este ahora también usará la librería “threading”.

```
import socket
import threading
```

Lo primero será crear la función para recibir mensajes del cliente. Esta recibe y decodifica hasta 1024 bytes, revisa si el mensaje dice “bye”, lo cual terminaría la conversación e imprimiría un mensaje diciendo lo mismo, en caso contrario, imprimiría “Cliente: “ y el mensaje recibido.

```
def recibir_mensajes(conexion):
    while True:
        try:
            mensaje = conexion.recv(1024).decode('utf-8') # Recibe hasta 1024 bytes y decodifica
            if mensaje.lower() == "bye":                  # Si el cliente dice "bye", termina el chat
                print("El cliente terminó la conversación.")
                break
            print(f"Cliente: {mensaje}")                  # Muestra el mensaje recibido
        except:
            break # Si ocurre un error (por ejemplo, desconexión), termina
    conexion.close()
```

Igualmente se creará la función para enviar mensajes al cliente, en esta se pide el mensaje a enviar, lo envía y termina el chat si se escribe “bye”.

```
def enviar_mensajes(conexion):
    while True:
        mensaje = input("Tú: ")                          # Pide entrada del servidor
        conexion.send(mensaje.encode('utf-8'))           # Envía el mensaje codificado
        if mensaje.lower() == "bye":                     # Si se escribe "bye", se termina el chat
            break
```

```
conexion.close()
```

Se creará la función principal “main”, en esta:

- Se crea una variable para el puerto
- Se crea el socket del servidor
- Vincula el socket a la IP local y al puerto especificado
- Escucha una conexión entrante
- Imprime “Servidor esperando conexión”
- Acepta la conexión del cliente
- Imprime “Cliente conectado desde” y la dirección
- Crea dos hilos para recibir y enviar mensajes, de igual manera los inicia y espera a que terminen
- Cierra el socket e imprime “Chat finalizado”

```
def main():  
    puerto = 4545  
    # Crea el socket del servidor  
    mi_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
    mi_socket.bind(("127.0.0.1",puerto))    # Lo vincula a la IP local y puerto 4545  
    mi_socket.listen(1)                    # Escucha una conexión entrante  
  
    print("Servidor esperando conexión...")  
    conexion, direccion = mi_socket.accept()    # Acepta conexión del cliente  
    print(f"Cliente conectado desde {direccion}")  
  
    # Crea dos hilos: uno para recibir y otro para enviar mensajes  
    hilo_receptor = threading.Thread(target=recibir_mensajes, args=(conexion,))  
    hilo_emisor = threading.Thread(target=enviar_mensajes, args=(conexion,))  
  
    # Inicia los hilos  
    hilo_receptor.start()  
    hilo_emisor.start()  
  
    # Espera a que ambos hilos terminen  
    hilo_receptor.join()  
    hilo_emisor.join()  
    mi_socket.close()  
    print("Chat finalizado.")
```

Finalmente se ejecuta la función principal

```
# Ejecuta la función principal
if __name__ == "__main__":
    main()
```

Ahora, para modificar el archivo de cliente.py, lo primero será también añadirle la librería “threading”

```
import socket
import threading
```

Se definirá la función para recibir mensajes del servidor, esta es una copia de la que se generó en el servidor

```
def recibir_mensajes(socket_cliente):
    while True:
        try:
            mensaje = socket_cliente.recv(1024).decode('utf-8') # Recibe y decodifica mensaje
            if mensaje.lower() == "bye": # Si recibe "bye", termina el chat
                print("El servidor terminó la conversación.")
                break
            print(f'Servidor: {mensaje}') # Muestra mensaje recibido
        except:
            break
    socket_cliente.close()
```

Se define la función para enviar mensajes al servidor, que igual será una copia de la del servidor.

```
def enviar_mensajes(socket_cliente):
    while True:
        mensaje = input("Tú: ") # Captura mensaje desde terminal
        socket_cliente.send(mensaje.encode('utf-8')) # Lo envía al servidor
        if mensaje.lower() == "bye": # Si escribes "bye", termina el chat
            break
    socket_cliente.close()
```

La principal diferencia está en la conexión que se genera en la función “main”. En el caso del cliente:

- Se guarda el nombre del host local en una variable

- Se guarda el puerto del servidor en una variable
- Se crea el socket del cliente
- Se conecta al servidor e imprime “Conectado al servidor. Puedes comenzar a chatear”.

Todo lo demás es igual al servidor (creación, inicialización y finalización de hilos).

```
def main():
    host = socket.gethostname() # Usa el nombre del host local
    puerto = 4545                # Mismo puerto que el servidor

    # Crea el socket del cliente
    socket_cliente = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    socket_cliente.connect((host, puerto))          # Conecta al servidor

    print("Conectado al servidor. Puedes comenzar a chatear.")

    # Crea dos hilos: uno para recibir, otro para enviar
    hilo_receptor = threading.Thread(target=recibir_mensajes, args=(socket_cliente,))
    hilo_emisor = threading.Thread(target=enviar_mensajes, args=(socket_cliente,))

    # Inicia ambos hilos
    hilo_receptor.start()
    hilo_emisor.start()

    # Espera a que ambos terminen
    hilo_receptor.join()
    hilo_emisor.join()
    print("Chat finalizado.")
```

Finalmente se ejecuta la función main.

```
if __name__ == "__main__":
    main()
```

A continuación se observa el correcto funcionamiento del lado del cliente

```
Unidad 7 — -zsh — 80x24
U7_A7_LFIB.docx
cliente.py
codigo1.py
codigo2.py
codigo3.py
codigo4.py
file1.txt
file2.txt
file3.txt
rpccliente.py
rpcserver.py
server.py
~$ A7_LFIB.docx
(base) izluis@MacBook-Pro-de-Campeon Unidad 7 % python3 cliente.py
Conectado al servidor. Puedes comenzar a chatear.
Tú: Hola
Tú: Servidor: Hola
¿Cómo estás?
Tú: Servidor: Bien y tu?
Bien, gracias!
Tú: Servidor: Ya me voy
Bye
Chat finalizado.
(base) izluis@MacBook-Pro-de-Campeon Unidad 7 %
```

Y el correcto funcionamiento del lado del servidor

```
(.conda) (base) izluis@MacBook-Pro-de-Campeon Maestria % /U
s/Maestria/Cómputo de Alto Rendimiento/Unidad 7/server.py"
Servidor esperando conexión...
Cliente conectado desde ('127.0.0.1', 50638)
Tú: Cliente: Hola
Hola
Tú: Cliente: ¿Cómo estás?
Bien y tu?
Tú: Cliente: Bien, gracias!
Ya me voy
Tú: El cliente terminó la conversación.
```

Podría mejorarse el display de los textos, pero en rasgos generales, funcionan correctamente.



## Ampliación código RPC

El objetivo de este ejercicio es tomar los códigos de “rpccliente.py” y “rpcserver.py” para añadir 5 nuevas funciones al servidor que se puedan llamar desde el cliente. El servidor deberá estar en una maquina distinta.

El código de “rpcserver.py” original solamente: Define una función suma, que toma x e y para sumarlos.

- Crea un servidor XML-RPC en el puerto 4242
- Registra la función con el nombre de “suma”
- Escucha peticiones indefinidamente

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

def suma(x,y):
    return x+y

servidor = SimpleXMLRPCServer(("", 4242))
print "Escuchando por el puerto 4242"
servidor.register_function(suma, 'suma')
servidor.serve_forever()
```

El código “rpccliente.py” original:

- Se conecta al servidor RPC en localhost:4242
- Llama la función remota suma con los valores 2 y 6 e imprime el resultado

```
from xmlrpclib import ServerProxy

servidor = ServerProxy("http://localhost:4242")
print servidor.suma(2,6)
```

Para la actualización de “rpcserver.py” se cambiaron las librerías para Python 3 y también se importó “math” para hacer las nuevas funciones.

```
from xmlrpc.server import SimpleXMLRPCServer
import math
```

Se crearon las nuevas funciones que estarán disponibles remotamente

```
def suma(x, y):  
    return x + y  
  
def resta(x, y):  
    return x - y  
  
def multiplica(x, y):  
    return x * y  
  
def divide(x, y):  
    if y == 0:  
        return "Error: División entre cero"  
    return x / y  
  
def potencia(x, y):  
    return x ** y  
  
def factorial(n):  
    if n < 0:  
        return "Error: Factorial no definido para negativos"  
    return math.factorial(n)
```

Se crea el servidor en localhost y en el puerto 8000

```
servidor = SimpleXMLRPCServer(("localhost", 8000), allow_none=True)  
print("Servidor RPC escuchando en el puerto 8000...")
```

Se registran todas las funciones

```
servidor.register_function(suma, 'suma')  
servidor.register_function(resta, 'resta')  
servidor.register_function(multiplica, 'multiplica')  
servidor.register_function(divide, 'divide')  
servidor.register_function(potencia, 'potencia')  
servidor.register_function(factorial, 'factorial')
```

Se inicia el servidor indefinidamente

```
servidor.serve_forever()
```

Para mejorar el código de "rpccliente.py" solamente se usa una librería compatible con Python 3

```
from xmlrpc.client import ServerProxy
```

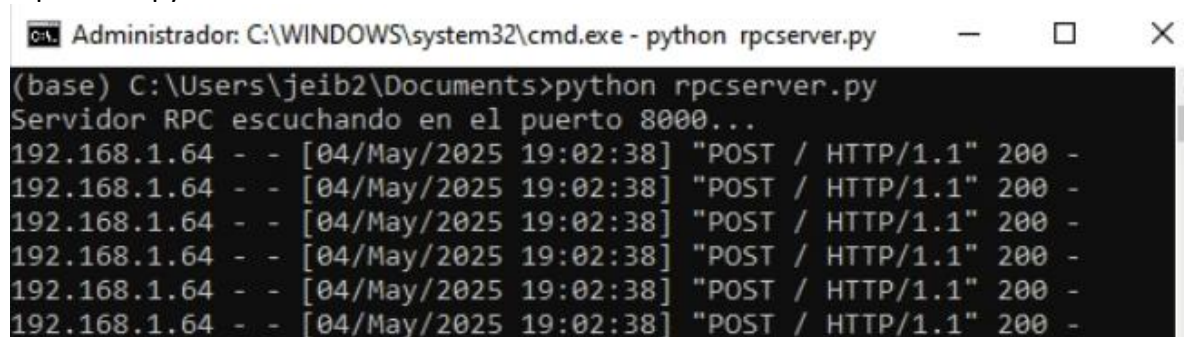
Se establece la conexión al servidor, en mi caso, utilizaré una laptop que se que está en la IP local 192.168.1.67 y el puerto será el 8000 (como se vió en el código del servidor)

```
servidor = ServerProxy("http://192.168.1.67:8000/")
```

Se prueban las funciones remotas

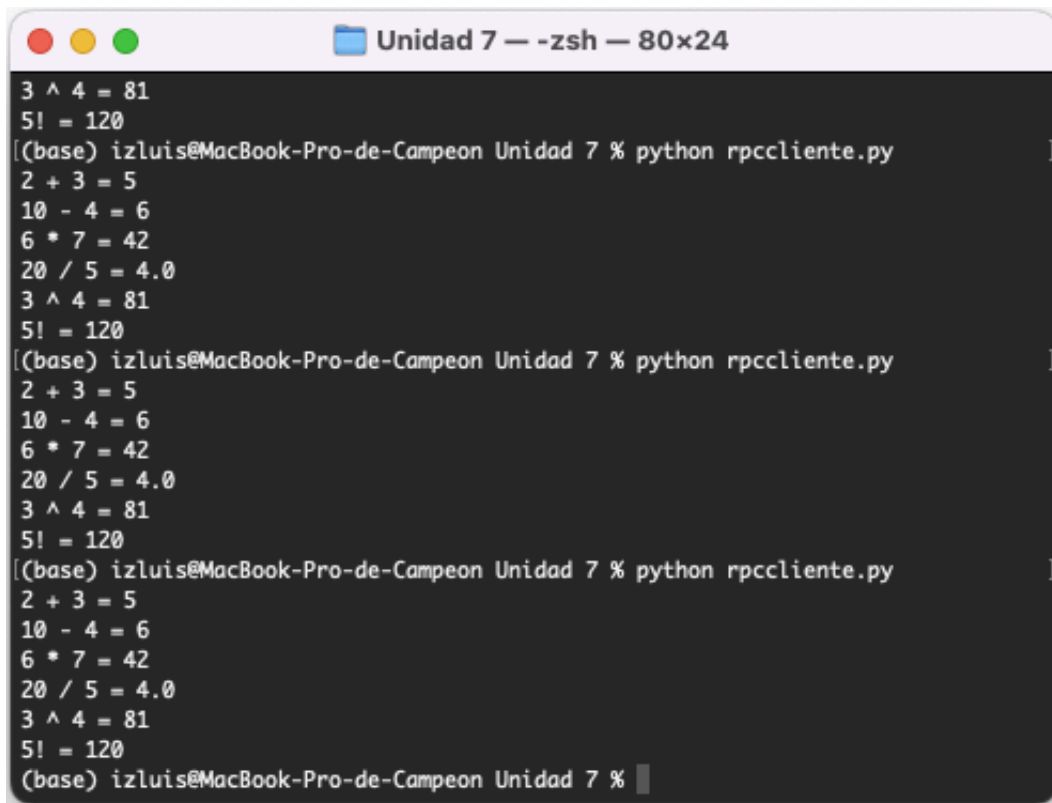
```
print("2 + 3 =", servidor.suma(2, 3))
print("10 - 4 =", servidor.resta(10, 4))
print("6 * 7 =", servidor.multiplica(6, 7))
print("20 / 5 =", servidor.divide(20, 5))
print("3 ^ 4 =", servidor.potencia(3, 4))
print("5! =", servidor.factorial(5))
```

En una computadora con Windows, en un ambiente de anaconda, se ejecutó el código "rpcserver.py"



```
Administrador: C:\WINDOWS\system32\cmd.exe - python rpcserver.py
(base) C:\Users\jeib2\Documents>python rpcserver.py
Servidor RPC escuchando en el puerto 8000...
192.168.1.64 - - [04/May/2025 19:02:38] "POST / HTTP/1.1" 200 -
192.168.1.64 - - [04/May/2025 19:02:38] "POST / HTTP/1.1" 200 -
192.168.1.64 - - [04/May/2025 19:02:38] "POST / HTTP/1.1" 200 -
192.168.1.64 - - [04/May/2025 19:02:38] "POST / HTTP/1.1" 200 -
192.168.1.64 - - [04/May/2025 19:02:38] "POST / HTTP/1.1" 200 -
192.168.1.64 - - [04/May/2025 19:02:38] "POST / HTTP/1.1" 200 -
```

En una computadora con MacOS se ejecutó el código "rpccliente.py"

A terminal window titled 'Unidad 7 - zsh - 80x24' showing three consecutive executions of a Python script named 'rpccliente.py'. Each execution outputs the same set of mathematical operations: 3 to the power of 4 equals 81, 5 factorial equals 120, 2 plus 3 equals 5, 10 minus 4 equals 6, 6 times 7 equals 42, and 20 divided by 5 equals 4.0. The prompt indicates the user is 'izluis@MacBook-Pro-de-Campeon' in the 'Unidad 7' directory.

```
Unidad 7 - zsh - 80x24
3 ^ 4 = 81
5! = 120
(base) izluis@MacBook-Pro-de-Campeon Unidad 7 % python rpccliente.py
2 + 3 = 5
10 - 4 = 6
6 * 7 = 42
20 / 5 = 4.0
3 ^ 4 = 81
5! = 120
(base) izluis@MacBook-Pro-de-Campeon Unidad 7 % python rpccliente.py
2 + 3 = 5
10 - 4 = 6
6 * 7 = 42
20 / 5 = 4.0
3 ^ 4 = 81
5! = 120
(base) izluis@MacBook-Pro-de-Campeon Unidad 7 % python rpccliente.py
2 + 3 = 5
10 - 4 = 6
6 * 7 = 42
20 / 5 = 4.0
3 ^ 4 = 81
5! = 120
(base) izluis@MacBook-Pro-de-Campeon Unidad 7 %
```

Como se puede observar, la comunicación fue exitosa y se ejecutaron todas las operaciones añadidas.

## Algoritmo de los filósofos comensales

El objetivo es diseñar un algoritmo en pseudocódigo para resolver el problema de los filósofos comensales. Tratando sobretodo evitar starvation y deadlock.

El problema de los filósofos comensales dice que 5 filósofos están sentados alrededor de una mesa circular y frente a cada uno de ellos se encuentra un plato de arroz y un palillo a la izquierda. Cada uno de los filósofos tiene que tomar dos palillos para poder comer y solamente hay 5 palillos en total (uno por cada uno de ellos), por lo cual solamente podrán comer 2 a la vez y los otros 3 deberán esperar.

Los problemas a evitar son:

- Deadlock: este ocurre si todos los filósofos toman el palillo de su izquierda al mismo tiempo y esperan para poder tomar el de la derecha, en este caso, ninguno puede continuar.
- Starvation: Ocurre cuando un filósofo no puede obtener nunca ambos palillos porque siempre se le adelantan los demás.

La solución a implementar es una cola First In, First Out con dos lock, uno serán los dos palillos a tomar y el otro se utiliza para acceder a la cola, igual se implementará un

semáforo para que solamente 2 filósofos puedan comer simultáneamente. Los filósofos entran en una cola por orden de llegada, solamente pueden comer cuando son los primeros en la cola y hay permisos disponibles para comer (máximo 2 permisos simultáneos), después de esto toman dos palillos y comen. Con esto se evita el deadlock porque aseguramos que solamente dos comen a la vez y también starvation porque todos terminan comiendo eventualmente.

El código se implementó en Python para corroborar que funcione, se utilizaron las librerías de "threading", "time" y "random".

```
import threading
import time
import random
```

Se añade el número de filósofos y palillos

```
N = 5
```

Cada palillo es un Lock entre filósofos

```
palillos = [threading.Lock() for _ in range(N)]
```

Se genera un lock para acceder a la cola de forma segura

```
lock_cola = threading.Lock()
```

Se genera el semáforo que actuará como permiso para comer y solo permitirá dos filósofos comiendo simultáneamente.

```
permiso_para_comer = threading.Semaphore(2)
```

En una función de filósofo tenemos varios pasos:

```
def filosofo(i):
    while True:
```

El primero será anadirse a la cola de espera

```
    with lock_cola:
        cola.append(i)
        print(f"Filósofo {i} se unió a la cola: {cola}")
```

Después espera a que sea su turno y haya permiso para comer, entonces avanza, caso contrario, vuelve a revisar en 0.1 segundos.

```
while True:
    with lock_cola:
        es_mi_turno = (cola[0] == i)
    if es_mi_turno and permiso_para_comer.acquire(blocking=False):
        break
    time.sleep(0.1)
```

El tercer paso es tomar dos palillos y comer (se le puso un sleep con tiempo aleatorio para simular comer)

```
palillos[i].acquire()
palillos[(i + 1) % N].acquire()

print(f"Filósofo {i} está comiendo...")
time.sleep(random.uniform(1, 2)) # Simula tiempo comiendo
```

Lo siguiente será liberar los palillos y el permiso para comer

```
palillos[i].release()
palillos[(i + 1) % N].release()
permiso_para_comer.release()
```

Finalmente termina de comer.

```
with lock_cola:
    print(f"Filósofo {i} terminó de comer.")
    cola.pop(0)
```

Lo siguiente será crear e iniciar los hilos

```
for i in range(N):
    threading.Thread(target=filosofo, args=(i,), daemon=True).start()
```

Al ejecutar el Código, se observa que todos los filósofos entran a la cola y ninguno se queda sin comer.

```

❶ (.conda) (base) izluis@MacBook-Pro-de-Campeon Maestria % /Users/izluis/Documents/Maestria/.conda/bin/python "/User
s/izluis/Documents/Maestria/C  puto de Alto Rendimiento/Unidad 7/codigo5.py"
Fil  sofo 0 se uni   a la cola: [0]
Fil  sofo 0 est   comiendo...
Fil  sofo 1 se uni   a la cola: [0, 1]
Fil  sofo 2 se uni   a la cola: [0, 1, 2]
Fil  sofo 3 se uni   a la cola: [0, 1, 2, 3]
Fil  sofo 4 se uni   a la cola: [0, 1, 2, 3, 4]
Fil  sofo 0 termin   de comer.
Fil  sofo 0 se uni   a la cola: [1, 2, 3, 4, 0]
Fil  sofo 1 est   comiendo...
Fil  sofo 1 termin   de comer.
Fil  sofo 1 se uni   a la cola: [2, 3, 4, 0, 1]
Fil  sofo 2 est   comiendo...
Fil  sofo 2 termin   de comer.
Fil  sofo 2 se uni   a la cola: [3, 4, 0, 1, 2]
Fil  sofo 3 est   comiendo...
Fil  sofo 3 termin   de comer.
Fil  sofo 3 se uni   a la cola: [4, 0, 1, 2, 3]
Fil  sofo 4 est   comiendo...
Fil  sofo 4 termin   de comer.
Fil  sofo 4 se uni   a la cola: [0, 1, 2, 3, 4]
Fil  sofo 0 est   comiendo...
Fil  sofo 0 termin   de comer.
Fil  sofo 0 se uni   a la cola: [1, 2, 3, 4, 0]
Fil  sofo 1 est   comiendo...
Fil  sofo 1 termin   de comer.
Fil  sofo 1 se uni   a la cola: [2, 3, 4, 0, 1]
Fil  sofo 2 est   comiendo...
Fil  sofo 2 termin   de comer.
Fil  sofo 2 se uni   a la cola: [3, 4, 0, 1, 2]

```

Al implementar esta soluci  n, parec  a ser una opci  n bastante correcta y, al observar los resultados se ve que no hay deadlock ni starvation, sin embargo, analiz  ndolos detenidamente, parec  a que es un proceso un poco m  s secuencial que paralelo. Estar  a interesante probar otras soluciones para lograr que sea m  s paralelo de lo que se observa actualmente.

## Investigaci  n sobre algoritmos de exclusi  n mutua.

### Algoritmo de Dekker

Este algoritmo fue una de las primeras soluciones al problema de exclusi  n mutua (mutex) entre dos procesos concurrentes. Este algoritmo permite que solamente un proceso entre a su secci  n cr  tica y el otro se queda esperando.

El algoritmo usa banderas booleanas para indicar si un proceso entr   a la secci  n cr  tica, usa una variable de turno y decide qui  n tiene la prioridad, en el caso de que ambos procesos intenten entrar, el turno se respeta hasta que el otro proceso lo ceda.

### Algoritmo de Peterson

Este algoritmo, como el de Dekker, tambi  n resuelve la exclusi  n mutua entre dos procesos, sin embargo, lo hace de una manera m  s sencilla.

Cada proceso indica que quiere entrar con una bandera booleana, al terminar se cede el turno al otro proceso, el cual solo entra si el otro no quiere o le ceden el turno.

## Bibliograf  a

*Unidad 4: Exclusión mutua, condiciones de carrera y sincronización.* (2017). [Presentación en PDF]. Benemérita Universidad Autónoma de Puebla, Facultad de Ciencias de la Computación. [https://www.cs.buap.mx/~hilario\\_sm/slide/pcp2016/unidad%204%20pcp-2017.pdf](https://www.cs.buap.mx/~hilario_sm/slide/pcp2016/unidad%204%20pcp-2017.pdf)

Arellano Vazquez, M. (s.f.). *Procesos*. INFOTEC.

Arellano Vazquez, M. (s.f.). *Comunicación entre procesos*. INFOTEC.