

Práctica Calificada: Implementación del Algoritmo de Lamport

Diana Yuliza Mamani vilca

17 de diciembre de 2024

Objetivo

El objetivo de esta práctica es implementar y analizar el funcionamiento del algoritmo de Lamport en un caso práctico de control de acceso concurrente, garantizando el orden lógico de los eventos en un sistema distribuido. Este experimento permite profundizar en la comprensión del uso de relojes lógicos para coordinar procesos y asegurar la consistencia en un entorno donde varios procesos trabajan simultáneamente sobre un mismo recurso compartido.

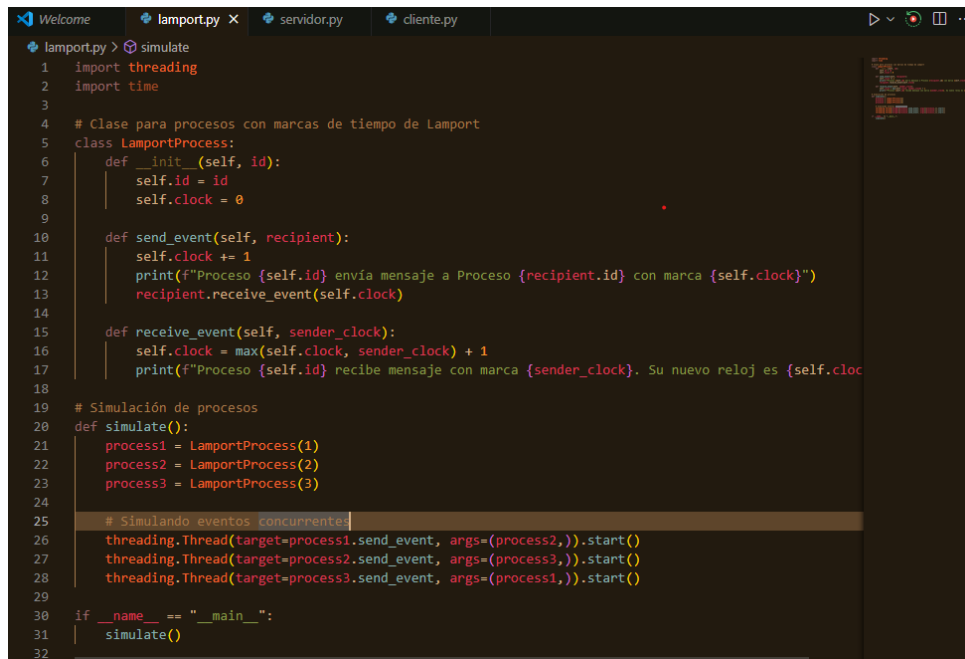
Caso Práctico

En este caso práctico, se simula un entorno compuesto por tres procesos distribuidos que intentan acceder de manera concurrente a un archivo compartido. La situación emula una problemática frecuente en sistemas distribuidos, donde múltiples procesos buscan escribir datos en un recurso compartido, y se debe evitar cualquier tipo de conflicto. Para solucionar esto y garantizar que las operaciones de escritura ocurran en un orden lógico y consistente, se utiliza el algoritmo de Lamport. Este algoritmo emplea relojes lógicos y mensajes entre los procesos para coordinar el acceso al recurso y garantizar que el orden de las operaciones refleje la secuencia correcta de eventos.

Implementación

La implementación del algoritmo se realizó utilizando el lenguaje de programación Python, debido a su simplicidad y amplia utilización en simulaciones de sistemas distribuidos. El lenguaje permite gestionar hilos y mecanismos de sincronización con facilidad, lo que resulta ideal para este caso práctico. A continuación, se presenta el fragmento principal del código utilizado para simular los procesos, sus interacciones y la escritura controlada en el archivo compartido.

Figura del Algoritmo



```
1 import threading
2 import time
3
4 # Clase para procesos con marcas de tiempo de Lamport
5 class LamportProcess:
6     def __init__(self, id):
7         self.id = id
8         self.clock = 0
9
10    def send_event(self, recipient):
11        self.clock += 1
12        print(f"Proceso {self.id} envía mensaje a Proceso {recipient.id} con marca {self.clock}")
13        recipient.receive_event(self.clock)
14
15    def receive_event(self, sender_clock):
16        self.clock = max(self.clock, sender_clock) + 1
17        print(f"Proceso {self.id} recibe mensaje con marca {sender_clock}. Su nuevo reloj es {self.clock}")
18
19 # Simulación de procesos
20 def simulate():
21     process1 = LamportProcess(1)
22     process2 = LamportProcess(2)
23     process3 = LamportProcess(3)
24
25     # Simulando eventos concurrentes
26     threading.Thread(target=process1.send_event, args=(process2,)).start()
27     threading.Thread(target=process2.send_event, args=(process3,)).start()
28     threading.Thread(target=process3.send_event, args=(process1,)).start()
29
30 if __name__ == "__main__":
31     simulate()
32
```

Figura 1: Porcentaje de algoritmos de consenso.

La figura presentada anteriormente ilustra una distribución general de la aplicación de algoritmos de consenso, donde el algoritmo de Lamport ocupa un lugar relevante en el control de concurrencia.

Resultados

La simulación llevada a cabo permitió observar cómo los procesos, a través de mensajes y relojes lógicos, coordinan sus acciones y logran acceder al archivo compartido de manera ordenada. Los resultados muestran que el algoritmo de Lamport garantiza que cada escritura ocurra de acuerdo con el orden lógico definido por los relojes. Además, se registraron capturas de pantalla que evidencian el funcionamiento correcto de la implementación.

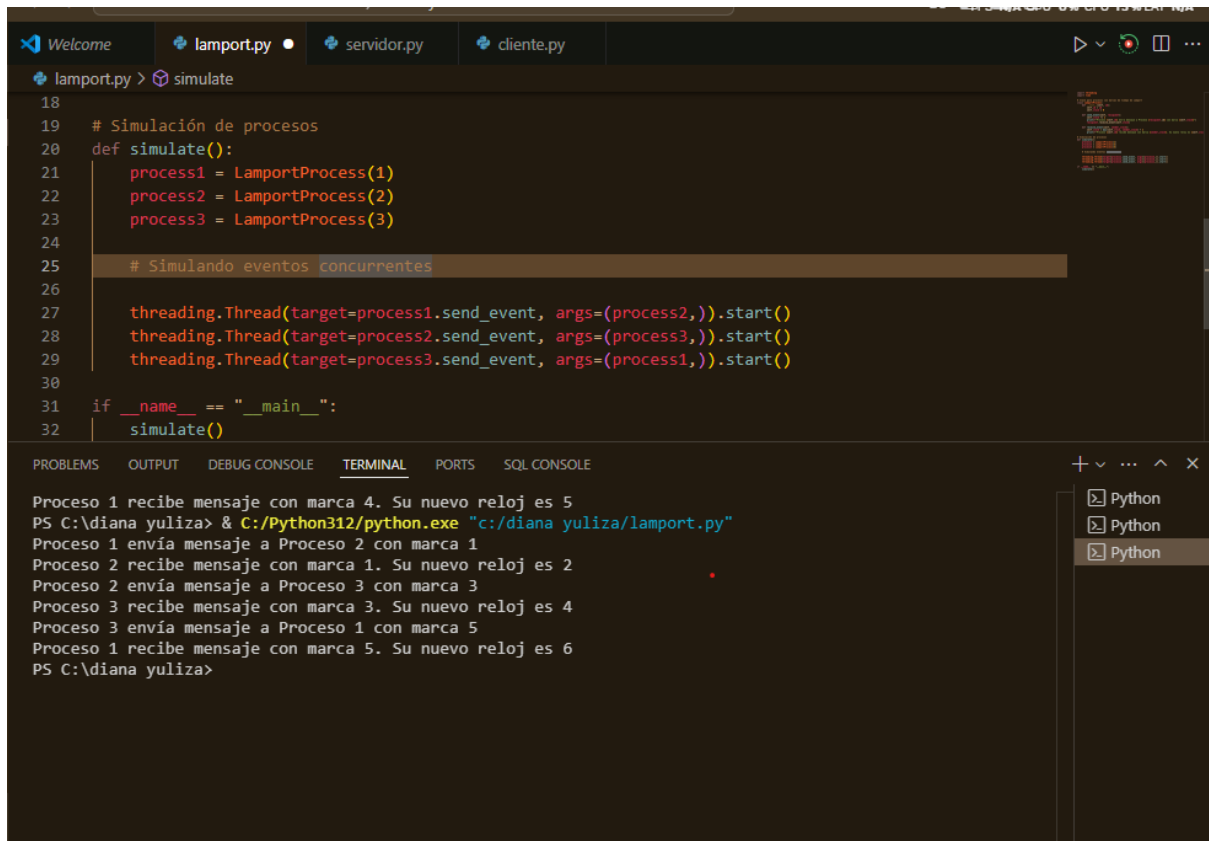
A continuación, se muestran las imágenes obtenidas durante la ejecución del programa:

La figura anterior muestra el momento en que el Proceso 1 obtiene acceso al recurso compartido y realiza la operación de escritura. Esta acción se realiza respetando el orden lógico definido por el reloj lógico.

La segunda figura presenta la secuencia completa de escritura realizada por los tres procesos. Se puede observar que el algoritmo logró sincronizar las operaciones correctamente y garantizó un acceso ordenado al archivo compartido.

Conclusión

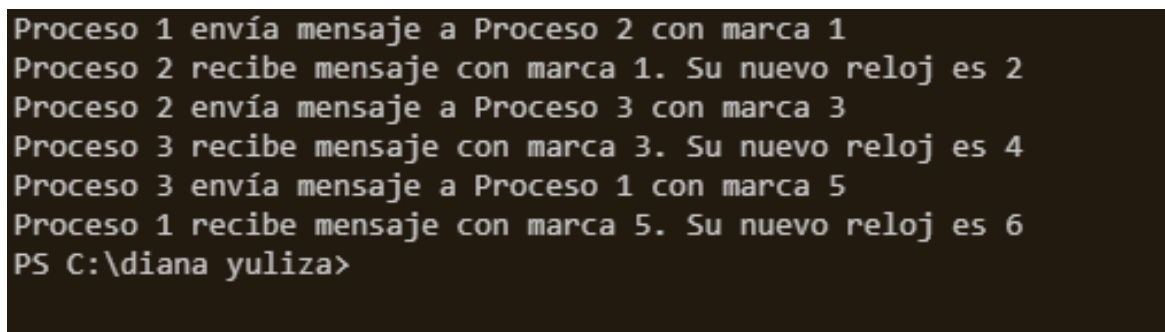
El algoritmo de Lamport demostró ser efectivo para coordinar accesos concurrentes en sistemas distribuidos. Durante la simulación, se logró garantizar un orden lógico en



```
18
19 # Simulación de procesos
20 def simulate():
21     process1 = LamportProcess(1)
22     process2 = LamportProcess(2)
23     process3 = LamportProcess(3)
24
25     # Simulando eventos concurrentes
26
27     threading.Thread(target=process1.send_event, args=(process2,)).start()
28     threading.Thread(target=process2.send_event, args=(process3,)).start()
29     threading.Thread(target=process3.send_event, args=(process1,)).start()
30
31 if __name__ == "__main__":
32     simulate()
```

Proceso 1 recibe mensaje con marca 4. Su nuevo reloj es 5
PS C:\diana yuliza> & C:/Python312/python.exe "c:/diana yuliza/lamport.py"
Proceso 1 envía mensaje a Proceso 2 con marca 1
Proceso 2 recibe mensaje con marca 1. Su nuevo reloj es 2
Proceso 2 envía mensaje a Proceso 3 con marca 3
Proceso 3 recibe mensaje con marca 3. Su nuevo reloj es 4
Proceso 3 envía mensaje a Proceso 1 con marca 5
Proceso 1 recibe mensaje con marca 5. Su nuevo reloj es 6
PS C:\diana yuliza>

Figura 2: Salida del proceso 1 escribiendo en el archivo.



```
Proceso 1 envía mensaje a Proceso 2 con marca 1
Proceso 2 recibe mensaje con marca 1. Su nuevo reloj es 2
Proceso 2 envía mensaje a Proceso 3 con marca 3
Proceso 3 recibe mensaje con marca 3. Su nuevo reloj es 4
Proceso 3 envía mensaje a Proceso 1 con marca 5
Proceso 1 recibe mensaje con marca 5. Su nuevo reloj es 6
PS C:\diana yuliza>
```

Figura 3: Secuencia completa de escritura de los procesos.

las operaciones de escritura realizadas por los tres procesos, a través del uso de relojes lógicos y la comunicación mediante mensajes.

Este ejercicio permitió comprender los fundamentos teóricos de los relojes lógicos y su aplicación práctica en la resolución de problemas de sincronización y control de concurrencia. Además, la implementación en Python proporcionó una base sólida para el desarrollo de simulaciones de sistemas distribuidos más complejos en el futuro.

1. Códigos del Proyecto

1.1. Código del servidor (servidor.py)

```

import threading
import time

class Proceso:
    def __init__(self, id):
        self.id = id
        self.reloj = 0

    def evento_interno(self):
        self.reloj += 1

    def enviar_mensaje(self, receptor):
        self.reloj += 1
        print(f"Proceso {self.id} env\u00eda solicitud a {receptor.id} con reloj {self.reloj}")
        receptor.recibir_mensaje(self.reloj, self.id)

    def recibir_mensaje(self, reloj_remitente, id_remitente):
        self.reloj = max(self.reloj, reloj_remitente) + 1
        print(f"Proceso {self.id} recibe mensaje de {id_remitente} -> Reloj actualizado: {self.reloj}")

    def escribir_archivo(self, recurso_compartido):
        with recurso_compartido:
            print(f"Proceso {self.id} est\u00e1 escribiendo en el archivo. Reloj l\u00f3gico: {self.reloj}")
            time.sleep(1) # Simula tiempo de escritura
            print(f"Proceso {self.id} termin\u00f3 de escribir en el archivo.")

# Simulaci\u00f3n de procesos
recurso_compartido = threading.Lock() # Simula el acceso al archivo

proceso1 = Proceso(1)
proceso2 = Proceso(2)
proceso3 = Proceso(3)

# Funciones para que los procesos interact\u00faen
def tarea_proceso1():
    proceso1.evento_interno()
    proceso1.enviar_mensaje(proceso2)
    proceso1.enviar_mensaje(proceso3)
    proceso1.escribir_archivo(recurso_compartido)

def tarea_proceso2():
    proceso2.evento_interno()
    proceso2.recibir_mensaje(proceso1.reloj, proceso1.id)
    proceso2.enviar_mensaje(proceso3)
    proceso2.escribir_archivo(recurso_compartido)

def tarea_proceso3():
    proceso3.recibir_mensaje(proceso1.reloj, proceso1.id)
    proceso3.recibir_mensaje(proceso2.reloj, proceso2.id)
    proceso3.evento_interno()
    proceso3.escribir_archivo(recurso_compartido)

# Hilos para simular concurrencia
hilo1 = threading.Thread(target=tarea_proceso1)

```

```
hilo2 = threading.Thread(target=tarea_proceso2)
hilo3 = threading.Thread(target=tarea_proceso3)

# Ejecutar los hilos
hilo1.start()
hilo2.start()
hilo3.start()

hilo1.join()
hilo2.join()
hilo3.join()

print("Simulaci\u00f3n completada.")
```

1.2. Código del cliente (cliente.py)

```
# Código del cliente para interactuar con el servidor
# (Vac\u00e9do por el momento: agregar implementaci\u00f3n si es necesario)
```