Proyecto 1. Una situación cotidiana con concurrencia y sincronización

1. Identificación y descripción del problema

1.1. Contexto personal y relevancia

En el análisis de mi día a día pude notar que, durante mis trayectos diarios a la universidad, pude observar la coordinación de intersecciones en las avenidas, la llegada inesperada de carros de y los pasos peatonales pueden provocar conflictos. Es por ello por lo que se vi la necesidad implementar un sistema de control (semáforos coordinados) para controlar el caos en la ciudad.

1.2. Problema por modelar

Simular una red urbana compuesta por varias intersecciones, las cuales son:

- Semáforos de control cíclico: Para las cuatro direcciones (norte, sur, este y oeste) con fases para vehículos y zonas exclusivas para pasos peatonales.
- Mecanismos de prevención: Dan prioridad a vehículos de emergencia (ambulancias, bomberos, etc.).
- Eventos externos: Solicitud de cruce peatonal, influyen en la sincronización de las luces.
- Rutas dinámicas: Los vehículos (hilos) siguen rutas que atraviesan varias intersecciones. Se modela la espera en semáforos y coordinación con otros.

1.3. Descripción de la situación que modelarán

- Intersecciones: Cada intersección es un objeto con su propio hilo de control de semáforos, donde se tiene
  - Semáforos multiterritorio: Con fases para carros, una o dos direcciones puede avanzar al mismo tiempo y fases para peatones.
  - Sensores y solicitudes: Variables compartidas que almacenan solicitudes del paso peatonal o la solicitad del paso de vehículos de emergencia.
- Vehículos: Cada vehículo se modela como un que determina su ruta en la red, se acerca a las intersecciones, consulta el estado del semáforo, y espera si la luz está en rojo. El vehículo puede ser normal o de emergencia.
- Peatones: Simulados como hilos que solicitan cruzar en las intersecciones.

1.4. Puntos de sincronización y problemas de concurrencia

Se debe garantizar que, durante el semáforo esté en verde los carros pueden avanzar en la dirección correcta o que la cantidad de carros concurrentes se limite de forma segura. Los carros y los peatones esperan hasta que el semáforo cambie de color. Además, la llegada de un vehículo de emergencia debe notificar a la red de semáforos para ajustar las secuencias de señales. Para evitar conflictos, las intersecciones vecinas pueden coordinar sus ciclos a través de un mecanismo de comunicación para asegurar un flujo continuo.

2. Implementación del modelo e introducción de mecanismo de sincronización

2.1.Código

```
import threading
import time
import random
import tkinter as tk
from tkinter import scrolledtext, messagebox
import queue
import math

# Constantes para los tiempos de ciclo de los semáforos (en segundos)
TIEMPO_VERDE = 6
TIEMPO_AMARILLO = 2
TIEMPO_PEATON = 4

# Cola para mensajes de registro y para comandos de dibujo
cola_registro = queue.Queue()
cola_dibujo = queue.Queue()

def registrar_mensaje(msg):
    timestamp = time.strftime("%H:%M:%S")
    cola_registro.put(f"[{timestamp}] {msg}\n")

# Coordenadas fijas para las intersecciones en el lienzo (x, y)
coordenadas_intersecciones = {
    "Intersección 1": (200, 200),
    "Intersección 2": (600, 200)
}

# Variable global para la comunicación entre intersecciones (simplificada)
estado_global_intersecciones = {}

# --------------------------
# LÓGICA DE LA SIMULACIÓN (HILOS)
# --------------------------

class Interseccion(threading.Thread):
    def __init__(self, nombre, vecinos=[]):
        super().__init__()
        self.nombre = nombre
        self.estado = "NS"  # Estado inicial: luz verde para Norte-Sur
        self.lock_interseccion = threading.Lock()
```

```
        # Se utiliza el lock en la condición para mayor seguridad
        self.cond_estado = threading.Condition(self.lock_interseccion)
        self.solicitud_peatonal = False
        self.emergencia = False
        self.vecinos = vecinos  # Nombres de intersecciones vecinas para coordinación
        estado_global_intersecciones[self.nombre] = self.estado

    def notificar_vecinos(self):
        global estado_global_intersecciones
        estado_global_intersecciones[self.nombre] = self.estado
        registrar_mensaje(f"[{self.nombre}] Estado actualizado a {self.estado}.
Vecinos: {self.vecinos}")
        # Enviar comando para actualizar la representación gráfica de la intersección
        cola_dibujo.put({"cmd": "actualizar_interseccion", "nombre": self.nombre,
"estado": self.estado})

    def ciclo_semaforo(self):
        with self.cond_estado:
            if self.emergencia:
                registrar_mensaje(f"[{self.nombre}] ¡MODO EMERGENCIA ACTIVADO!")
                self.estado = "EMERGENCIA"
                self.cond_estado.notify_all()
                self.notificar_vecinos()
                time.sleep(3)
                self.emergencia = False
            elif self.solicitud_peatonal:
                if self.estado == "NS":
                    self.estado = "PEATON_NS"
                else:
                    self.estado = "PEATON_EW"
                registrar_mensaje(f"[{self.nombre}] Fase peatonal activada:
{self.estado}")
                self.cond_estado.notify_all()
                self.notificar_vecinos()
                time.sleep(TIEMPO_PEATON)
                self.solicitud_peatonal = False

    def run(self):
        while True:
            with self.cond_estado:
                registrar_mensaje(f"[{self.nombre}] Fase {self.estado} activa. Vehí-
culos avanzan.")
                self.notificar_vecinos()
                self.cond_estado.notify_all()
            time.sleep(TIEMPO_VERDE)

            with self.cond_estado:
                registrar_mensaje(f"[{self.nombre}] Fase amarilla.")
                self.notificar_vecinos()
            time.sleep(TIEMPO_AMARILLO)

            self.ciclo_semaforo()

            with self.cond_estado:
                if self.estado == "NS":
                    self.estado = "EW"
                elif self.estado == "EW":
                    self.estado = "NS"
                elif self.estado in ("PEATON_NS", "PEATON_EW", "EMERGENCIA"):
                    self.estado = "NS"
                registrar_mensaje(f"[{self.nombre}] Cambiando a fase {self.estado}.")
```

```
                self.notificar_vecinos()
                self.cond_estado.notify_all()

class Vehiculo(threading.Thread):
    contador_vehiculos = 0

    def __init__(self, nombre, ruta, es_emergencia=False):
        super().__init__()
        self.nombre = nombre
        self.ruta = ruta  # Lista de intersecciones (objetos) que debe cruzar
        self.es_emergencia = es_emergencia
        self.id_grafico = None  # ID del objeto en el lienzo
        Vehiculo.contador_vehiculos += 1
        self.id_vehiculo = Vehiculo.contador_vehiculos

    def mover_grafico(self, inicio, fin):
        # Simula el movimiento animado del vehÃculo entre dos puntos
        pasos = 20
        x0, y0 = inicio
        x1, y1 = fin
        dx = (x1 - x0) / pasos
        dy = (y1 - y0) / pasos
        for i in range(pasos):
            nuevo_x = x0 + dx * (i + 1)
            nuevo_y = y0 + dy * (i + 1)
            cola_dibujo.put({"cmd": "mover_vehiculo", "id": self.id_vehiculo, "x":
nuevo_x, "y": nuevo_y})
            time.sleep(0.1)
        cola_dibujo.put({"cmd": "eliminar_vehiculo", "id": self.id_vehiculo})

    def run(self):
        for interseccion in self.ruta:
            tipo = "EMERGENCIA" if self.es_emergencia else "NORMAL"
            registrar_mensaje(f"Vehículo {self.nombre} ({tipo}) se aproxima a
{interseccion.nombre}.")
            with interseccion.cond_estado:
                if self.es_emergencia:
                    interseccion.emergencia = True
                    interseccion.cond_estado.notify_all()

                while True:
                    if interseccion.estado not in ("PEATON_NS", "PEATON_EW"):
                        break
                    registrar_mensaje(f"Vehículo {self.nombre} espera en
{interseccion.nombre} (estado: {interseccion.estado}).")
                    interseccion.cond_estado.wait(timeout=1)
            with interseccion.lock_interseccion:
                registrar_mensaje(f"Vehículo {self.nombre} está cruzando
{interseccion.nombre}.")
                coord_inicio = coordenadas_intersecciones[interseccion.nombre]
                destino = coord_inicio
                indice = self.ruta.index(interseccion)
                if indice + 1 < len(self.ruta):
                    destino = coordenadas_intersecciones[self.ruta[indice +
1].nombre]
                else:
                    destino = (coord_inicio[0] + 100, coord_inicio[1])
                cola_dibujo.put({
                    "cmd": "crear_vehiculo",
                    "id": self.id_vehiculo,
                    "nombre": self.nombre,
```

```
                    "x": coord_inicio[0],
                    "y": coord_inicio[1],
                    "color": "red" if self.es_emergencia else "blue"
                })
                self.mover_grafico(coord_inicio, destino)
                registrar_mensaje(f"Vehículo {self.nombre} ha cruzado
{interseccion.nombre}.")
            time.sleep(random.uniform(0.5, 2))
        registrar_mensaje(f"Vehículo {self.nombre} ha completado su ruta.")

class Peaton(threading.Thread):
    def __init__(self, nombre, interseccion):
        super().__init__()
        self.nombre = nombre
        self.interseccion = interseccion

    def run(self):
        registrar_mensaje(f"Peaton {self.nombre} solicita cruzar en
{self.interseccion.nombre}.")
        with self.interseccion.cond_estado:
            self.interseccion.solicitud_peatonal = True
            self.interseccion.cond_estado.notify_all()
        cola_dibujo.put({"cmd": "crear_peaton", "nombre": self.nombre,
"interseccion": self.interseccion.nombre, "color": "green"})
        time.sleep(TIEMPO_PEATON + random.uniform(0.5, 1.5))
        cola_dibujo.put({"cmd": "eliminar_peaton", "nombre": self.nombre})
        registrar_mensaje(f"Peaton {self.nombre} ha cruzado en
{self.interseccion.nombre}.")

def simulacion_principal():
    inter1 = Interseccion("Intersección 1", vecinos=["Intersección 2"])
    inter2 = Interseccion("Intersección 2", vecinos=["Intersección 1"])
    inter1.daemon = True
    inter2.daemon = True

    inter1.start()
    inter2.start()

    vehiculos = []
    for i in range(10):
        ruta = random.choice([[inter1, inter2], [inter2, inter1]])
        es_emergencia = random.random() < 0.2
        veh = Vehiculo(f"V{i+1}", ruta, es_emergencia)
        vehiculos.append(veh)
        time.sleep(random.uniform(0.3, 1))
        veh.start()

    peatones = []
    for i in range(5):
        interseccion = random.choice([inter1, inter2])
        p = Peaton(f"P{i+1}", interseccion)
        peatones.append(p)
        time.sleep(random.uniform(0.5, 1.5))
        p.start()

    for veh in vehiculos:
        veh.join()
    for p in peatones:
        p.join()
```

```
    registrar_mensaje("Simulación completada: Todos los vehículos y peatones han
cruzado.")

# ---------------------------
# INTERFAZ GRÁFICA (TKINTER)
# ---------------------------

class SimuladorTraficoGUI:
    def __init__(self, master):
        self.master = master
        master.title("Simulador de Tráfico en la ciudad")
        master.geometry("900x700")
        master.resizable(False, False)

        self.encabezado = tk.Label(master, text="Simulador de Tráfico en la ciudad",
font=("Helvetica", 20, "bold"))
        self.encabezado.pack(pady=10)
        self.explicacion = tk.Label(master, text="Este simulador modela una ciudad
con intersecciones coordinadas, vehículos (normales y de emergencia) y
peatones.\nPresione 'Iniciar Simulación' para comenzar.", wraplength=850,
justify="center")
        self.explicacion.pack(pady=5)

        self.boton_iniciar = tk.Button(master, text="Iniciar Simulación",
font=("Helvetica", 16), command=self.iniciar_simulacion)
        self.boton_iniciar.pack(pady=10)

        self.lienzo = tk.Canvas(master, width=850, height=350, bg="lightgray")
        self.lienzo.pack(pady=10)

        # Inicializar diccionarios de objetos gráficos antes de dibujar
        self.intersecciones_items = {}  # {nombre: id_objeto_canvas}
        self.vehiculos_items = {}       # {id_vehiculo: id_objeto_canvas}
        self.peatones_items = {}        # {nombre: id_objeto_canvas}

        self.dibujar_intersecciones()

        self.area_registro = scrolledtext.ScrolledText(master, width=105, height=15,
font=("Consolas", 10))
        self.area_registro.pack(pady=10)
        self.area_registro.config(state=tk.DISABLED)

        self.master.protocol("WM_DELETE_WINDOW", self.al_cerrar)

        self.actualizar_registro()
        self.actualizar_graficos()

    def dibujar_intersecciones(self):
        for nombre, (x, y) in coordenadas_intersecciones.items():
            tam = 80
            id_item = self.lienzo.create_rectangle(x - tam//2, y - tam//2, x +
tam//2, y + tam//2, fill="white", outline="black", width=2)
            self.lienzo.create_text(x, y, text=nombre, font=("Helvetica", 10,
"bold"))
            self.intersecciones_items[nombre] = id_item

    def iniciar_simulacion(self):
        self.boton_iniciar.config(state=tk.DISABLED)
        hilo_simulacion = threading.Thread(target=simulacion_principal, daemon=True)
        hilo_simulacion.start()
        registrar_mensaje("Simulación iniciada...")
```

```
    def actualizar_registro(self):
        while not cola_registro.empty():
            msg = cola_registro.get()
            self.area_registro.config(state=tk.NORMAL)
            self.area_registro.insert(tk.END, msg)
            self.area_registro.yview(tk.END)
            self.area_registro.config(state=tk.DISABLED)
        self.master.after(100, self.actualizar_registro)

    def actualizar_graficos(self):
        while not cola_dibujo.empty():
            cmd = cola_dibujo.get()
            if cmd["cmd"] == "actualizar_interseccion":
                nombre = cmd["nombre"]
                estado = cmd["estado"]
                self.actualizar_grafico_interseccion(nombre, estado)
            elif cmd["cmd"] == "crear_vehiculo":
                vid = cmd["id"]
                x = cmd["x"]
                y = cmd["y"]
                color = cmd["color"]
                id_objeto = self.lienzo.create_oval(x-10, y-10, x+10, y+10,
fill=color, outline="black", width=1)
                self.vehiculos_items[vid] = id_objeto
                # Vincular clic para mostrar informaciÃ³n
                self.lienzo.tag_bind(id_objeto, "<Button-1>", lambda event, id=vid:
self.al_clic_vehiculo(id))
            elif cmd["cmd"] == "mover_vehiculo":
                vid = cmd["id"]
                x = cmd["x"]
                y = cmd["y"]
                if vid in self.vehiculos_items:
                    self.lienzo.coords(self.vehiculos_items[vid], x-10, y-10, x+10,
y+10)
            elif cmd["cmd"] == "eliminar_vehiculo":
                vid = cmd["id"]
                if vid in self.vehiculos_items:
                    self.lienzo.delete(self.vehiculos_items[vid])
                    del self.vehiculos_items[vid]
            elif cmd["cmd"] == "crear_peaton":
                nombre = cmd["nombre"]
                interseccion = cmd["interseccion"]
                x, y = coordenadas_intersecciones[interseccion]
                id_objeto = self.lienzo.create_rectangle(x-5, y-5, x+5, y+5,
fill="green", outline="black", width=1)
                self.peatones_items[nombre] = id_objeto
            elif cmd["cmd"] == "eliminar_peaton":
                nombre = cmd["nombre"]
                if nombre in self.peatones_items:
                    self.lienzo.delete(self.peatones_items[nombre])
                    del self.peatones_items[nombre]
        self.master.after(10, self.actualizar_graficos)

    def actualizar_grafico_interseccion(self, nombre, estado):
        id_item = self.intersecciones_items.get(nombre)
        if id_item:
            if estado == "NS":
                self.lienzo.itemconfig(id_item, fill="green")
            elif estado == "EW":
                self.lienzo.itemconfig(id_item, fill="blue")
```

```
            elif estado == "PEATON_NS":
                self.lienzo.itemconfig(id_item, fill="yellow")
            elif estado == "PEATON_EW":
                self.lienzo.itemconfig(id_item, fill="orange")
            elif estado == "EMERGENCIA":
                self.lienzo.itemconfig(id_item, fill="red")
            else:
                self.lienzo.itemconfig(id_item, fill="white")

    def al_clic_vehiculo(self, id_vehiculo):
        messagebox.showinfo("Información del Vehículo", f"ID del Vehículo:
{id_vehiculo}")

    def al_cerrar(self):
        if messagebox.askyesno("Salir", "¿Desea salir de la simulación?"):
            self.master.destroy()

# ---------------------------
# FUNCION PRINCIPAL
# ---------------------------

def main():
    root = tk.Tk()
    app = SimuladorTraficoGUI(root)
    root.mainloop()

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

4. Documentación

4.1. Descripción de los mecanismos de sincronización empleados.

- **Locks y Semáforos:** Se usan para garantizar la exclusión mutua en la intersección y para limitar el número de vehículos que pueden cruzar simultáneamente en una misma dirección.
- **Variables de Condición (Condition Variables):** Permiten que vehículos y peatones esperen hasta que la intersección tenga la fase correspondiente. Además, permiten la notificación inmediata cuando se detecta la llegada de un vehículo de emergencia o la solicitud de cruce peatonal.
- **Eventos y Preempción:** Una variable de estado que indica si se ha detectado un vehículo de emergencia. Al activarse, se notifica a los hilos de control de semáforos para cambiar la fase, interrumpiendo el ciclo normal.

4.2. Lógica de operación

El hilo de cada intersección ejecuta un ciclo en el que se establecen fases para los vehículos (por ejemplo, "NS" y "EW"). Se revisan constantemente las solicitudes de eventos especiales. En caso de emergencia, se activa la fase "EMERGENCIA".

Si se solicita el cruce peatonal, se activa una fase exclusiva para peatones ("PEATON_NS" o "PEATON_EW"). Se notifica a todos los hilos asociados (vehículos y peatones) al producirse un cambio. Cada intersección actualiza un estado global (simulado en global_intersections_state) y notifica a sus vecinas. Esto permite que se puedan implementar estrategias de sincronización avanzada en una red real.

- **estado:** Variable que define la fase actual (por ejemplo, "NS", "EW", "EMERGENCIA", "PEATON_NS", "PEATON_EW").
- **solicitud_peatonal y emergencia:** Flags que indican la necesidad de cambiar la fase.
- **lock_interseccion y cond_estado:** Mecanismos de sincronización para coordinar el acceso y notificar cambios.
- **global_intersections_state:** Simula la comunicación entre intersecciones adyacentes para la coordinación de ciclos.

4.2.1. Descripción algorítmica del avance de cada hilo/proceso

1. Controlador de Intersección: Revisa solicitudes de cruce peatonal y la presencia de vehículos de emergencia. Ejecuta ciclos de semáforo para las direcciones (por ejemplo, verde para Norte-Sur, luego para Este-Oeste, intercalando fases peatonales). Si se detecta un evento prioritario, interrumpe el ciclo normal y activa la fase especial.
2. Vehículos: Cada vehículo elige su ruta y, al llegar a una intersección, consulta el estado del semáforo. Si la fase es la adecuada, adquiere el lock de la intersección y “cruza”.
3. Peatones: Los peatones se generan en momentos aleatorios y solicitan el cruce presionando un “botón virtual”. La intersección, al detectar la solicitud, reserva una fase exclusiva para peatones y notifica a los vehículos para que se detengan.
4. Coordinación entre Intersecciones: Cada intersección puede comunicar su estado a intersecciones adyacentes, lo que permite ajustar los tiempos de los semáforos en función del flujo detectado en las rutas principales.

4.3. Descripción del entorno del desarrollo, suficiente para reproducir una ejecución exitosa

4.3.1. Lenguaje Empleado

Lenguaje Python 3.8+

4.3.2. Bibliotecas

Threading: Para crear y gestionar hilos, locks, conditions y semáforos.

Time: Simular retardos y ciclos de semáforos.

Random: Para generar la aleatoriedad en la llegada de vehículos y peatones.

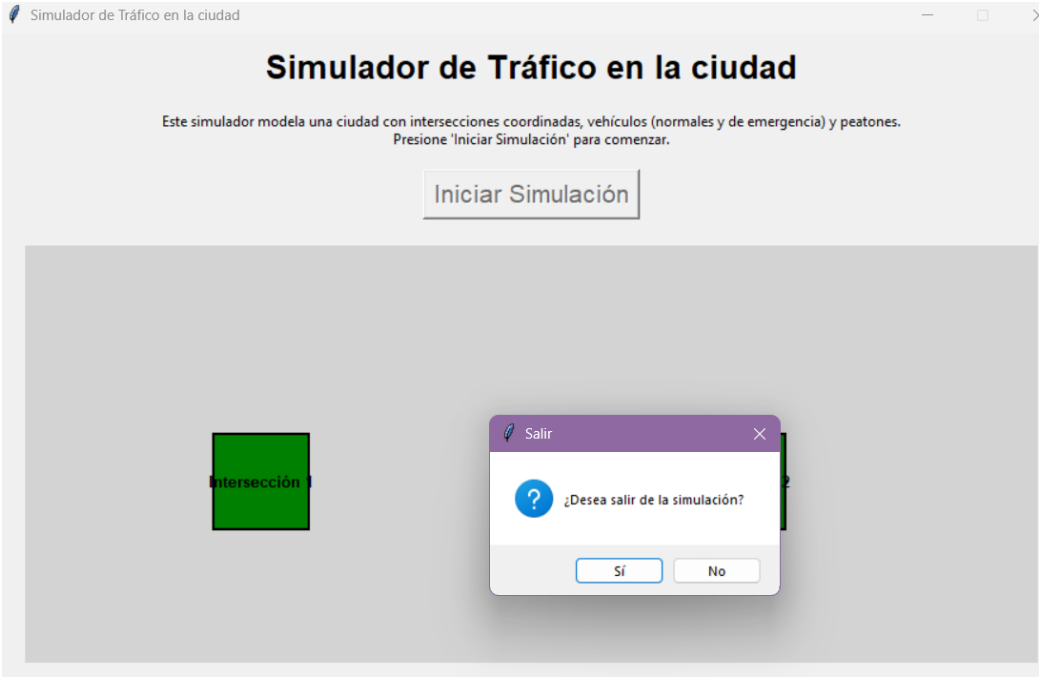
4.3.3. Editor/Herramienta: Se utilizó el IDE Visual Studio Code, donde se utilizó el lenguaje Python.

4.3.4. Ejecución y resultados:

Se abre la interfaz donde se presenta la simulación:

Se inicia la simulación**.**

Para salir de la simulación le damos en el tache.



5. Conclusiones

Este proyecto donde se simula una red urbana de tráfico con múltiples intersecciones y actores concurrentes. Se implementa un sistema de sincronización mediante locks, variables de condición, y flags de control para manejar eventos prioritarios permitiendo coordinar múltiples hilos en un entorno dinámico y realista.

La solución no solo aborda la sincronización a nivel local de cada intersección, sino que también establece un mecanismo básico de coordinación global, lo que demuestra la aplicación de concurrencia en un problema de la vida real.