

EASYCAB



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1																				
2																				
3																				
4																				
5																				
6																				
7																				
8																				
9																				
10																				
11																				
12																				
13																				
14																				
15																				
16																				
17																				
18																				
19																				
20																				

Estado general del sistema: OK

*** EASY CAB Release 1 ***																				
Taxis					Clientes															
Id.	Destino	Estado		Id.	Destino	Estado														
4	a	OK.	Servicio a	4	a	A														OK. Taxi 4
1	g	.	.	2	.	.														
2	.	.	.	3	.	.														
3	g	.	.	4	.	.														
4	.	.	.	5	A	.														
5	.	.	.	6	.	.														
6	.	.	.	7	.	.														
7	.	.	.	8	.	.														
8	.	.	.	9	.	.														
9	.	.	.	10	.	.														
10	.	.	.	11	.	.														
11	.	.	.	12	.	.														
12	.	.	.	13	.	.														
13	.	.	.	14	.	.														
14	.	.	.	15	.	.														
15	.	.	.	16	.	.														
16	.	.	.	17	.	.														
17	.	.	.	18	.	.														
18	.	.	.	19	.	.														
19	.	.	.	20	.	.														

Estado general del sistema: OK

UNIVERSIDAD DE ALICANTE - INGENIERÍA INFORMÁTICA 2024/2025

Nombre Estudiante 1: Adrián Rodríguez-Barbero Castillo
DNI Estudiante 1: 51236400J

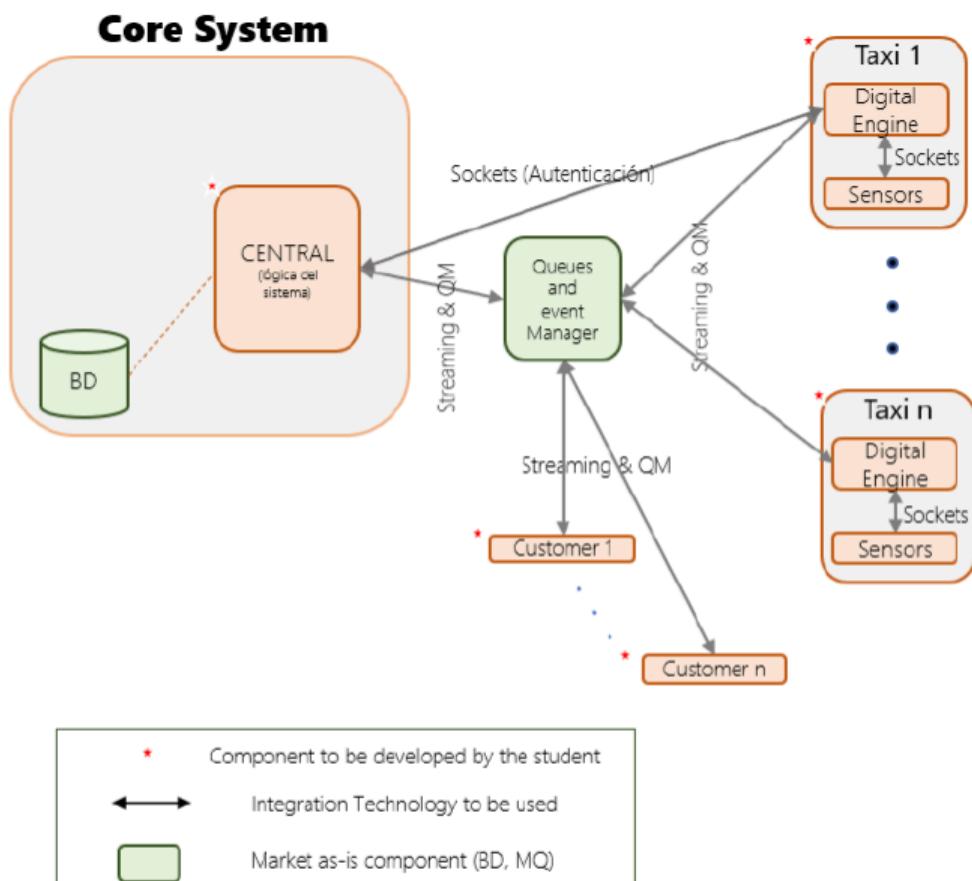
Nombre Estudiante 2: Javier García Cutillas
DNI Estudiante 2: 20082540K

ÍNDICE

Nombre y descripción de los componentes software desarrollados.

Guía de despliegue de la aplicación.

Capturas de pantalla mostrando el funcionamiento de la aplicación.



Nombre y descripción de los componentes software desarrollados.

Central -> EC_Central

bloque común en python, encargado de ejecutar el script principal y manejar los parámetros desde consola, es decir, desde la línea de comandos

```
if __name__ == "__main__":
    if (len(sys.argv) == 5):
        SOCKET_IP = sys.argv[1]
        KAFKA_IP = sys.argv[3]
        KAFKA_PORT = int(sys.argv[4])
        PRODUCER = kafka.KafkaProducer(bootstrap_servers=f'{KAFKA_IP}:{KAFKA_PORT}')

        main(sys.argv[2])

    else:
        print("ERROR! Se necesitan estos argumentos: <IP DE ESCUCHA> <PUERTO DE ESCUCHA> <IP DEL BOOTSTRAP-SERVER> <PUERTO DEL BOOTSTRAP-SERVER>")
```

Variables globales:

```
HEADER = 64
FORMAT = 'utf-8'
END_CONNECTION1 = "FIN"
END_CONNECTION2 = "ERROR"
KAFKA_IP = 0
KAFKA_PORT = 0
PRODUCER = 0
SOCKET_IP = 0

CHAR_MAP = "."
SPACE = " "
LINE = f"{'-' * 64}\n"

# PARA MOSTRAR MAPA
#####
mapa = [[["."] for _ in range(20)] for _ in range(20)]
taxis = []
customers = []
locations = []
mapa2Print = []
```

Todas estas son las variables globales del programa las cuales sirven o para mostrar el mapa, o información necesaria para Kafka o crear el productor (las que están en mayúscula). Las variables en minúscula son las que mantienen la funcionalidad completa del sistema, mapa almacena todo el mapa, taxis almacena los taxis activos en el sistema,

customers almacena todos los clientes activos del sistema, locations almacena todas las localizaciones del archivo EC_locations.json y mapa2Print simplemente es una lista para posteriormente mostrar y enviar el mapa a través de Kafka.

main()

```
# MAIN
def main(port):
    print("INICIANDO CENTRAL...")
    server = socket.gethostbyname(SOCKET_IP)

    addr = (server, int(port))

    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(addr)

    manageTaxi(CLEAN)

    threadSockets = threading.Thread(target=connectionSocket, args=(server,))
    threadSockets.start()

    threadRequests = threading.Thread(target=requestCustomers)
    threadRequests.start()

    threadAreActives = threading.Thread(target=areActives)
    threadAreActives.start()

    threadTaxiKeys = threading.Thread(target=taxiKeys)
    threadTaxiKeys.start()

    # PARA MOSTRAR MAPA
    #####
    loadLocations()
    mostrar_mapa()

    threadMovimientoTaxi = threading.Thread(target=receiveInfoTaxi)
    threadMovimientoTaxi.start()
    #####
    return 0
```

Es la función principal, inicializando la central a través de un puerto de escucha que recibe por parámetro, y configura varios hilos, para hacer la autenticación por socket con el taxi, para recibir información de los clientes, para ver si están activos tanto taxis como clientes, para detectar las teclas del teclado, y para recibir la información del taxi, donde se le dicen los movimientos para actualizar el mapa.

Esta aplicación incluye los siguientes módulos:

Comenzaré mostrando el módulo que controla los taxis desde la central.

No recibe nada por parámetros.

```

# Gestionar acciones desde el Central a un Taxi
def taxiKeys():
    while True:
        detectedKeys = detectKeys()

        if detectedKeys is not None:
            try:
                idTaxi = detectedKeys[0]

                if int(idTaxi) not in taxis:
                    print(f"NO EXISTE EL TAXI {idTaxi}")

            else:
                idTaxi = int(idTaxi)

                actionTaxiKey = detectedKeys[1]
                actionTaxi = "" # (idTaxi)p - Parar ### (idTaxi)r - Reanudar ### (idTaxi)d - Destino ### (idTaxi)b - Base
                if actionTaxiKey == "p":
                    actionTaxi = "PARAR"
                elif actionTaxiKey == "r":
                    actionTaxi = "REANUDAR"
                elif actionTaxiKey == "d":
                    destino = None
                    for location in locations:
                        if location[0] == detectedKeys[2]:
                            destino = detectedKeys[2]

                if destino is not None:
                    isConnected, _ = manageTaxi(CONNECT2TAXI_ID, idCustomer="Central", destination=detectedKeys[2], idTaxi=idTaxi)
                    if isConnected:
                        locationTuple = getLocation(destino)
                        x = locationTuple[0]
                        y = locationTuple[1]
                        sendMessageKafka("Central2Taxi", f"TAXI {idTaxi} DESTINO {x},{y} CLIENTE CENTRAL")

                    else:
                        print(f"NO EXISTE LA LOCALIZACIÓN {detectedKeys[2]}")

                elif actionTaxiKey == "b":
                    isConnected, _ = manageTaxi(CONNECT2TAXI_ID, idCustomer="Central", destination="Base", idTaxi=idTaxi)
                    if isConnected:
                        sendMessageKafka("Central2Taxi", f"TAXI {idTaxi} DESTINO 1,1 CLIENTE CENTRAL")

                sendMessageKafka("Central2Taxi", f"TAXI {idTaxi},CENTRAL {actionTaxi}")

            except Exception:
                print(f"NO EXISTE EL TAXI {idTaxi}")

            time.sleep(0.25)

```

En esta función se puede observar cómo se podría enviar un taxi a una localización en concreto, volver a base o incluso parar el taxi o reanudarlo. Esto se realizaría pulsando la tecla de la id del taxi y posteriormente su acción con su añadido de que si se pulsa d (para colocar un destino) se debe pulsar otra tecla para indicar a qué destino quiere ir.

El siguiente módulo tampoco recibe nada por parámetro y su funcionalidad es la de detectar las teclas de la función anterior:

```

# CLASES
# Detectar dos teclas o 3 teclas seguidas
def detectKeys():
    if msvcrt.kbhit():
        firstKey = msvcrt.getch().decode(FORMAT).lower()

    while True:
        if msvcrt.kbhit():
            secondKey = msvcrt.getch().decode(FORMAT).lower()

            if secondKey == "d":
                while True:
                    if msvcrt.kbhit():
                        thirdKey = msvcrt.getch().decode(FORMAT)
                        # Devolvemos las tres teclas concatenadas en el caso que tengamos que enviarlo a un destino
                        return firstKey + secondKey + thirdKey

            # Devolvemos las dos teclas concatenadas
            return firstKey + secondKey

```

Como se puede observar se debe introducir la primera pulsación, posteriormente la segunda y finalmente, y si la segunda es la d, se debe seleccionar la ubicación. Las siguiente funciones realizan la misma funcionalidad que es controlar la desconexión de un taxi o un cliente:

```
# GESTIONA EL MANTENIMIENTO DE LOS CLIENTES O LOS TAXIS
# Deshabilitar los taxis o clientes no activos
def disableNoActives(activeTaxis, activeCustomers):
    for taxi in taxis:
        if taxi not in activeTaxis:
            sendMessageKafka("Central2Taxi", f"FIN {taxi}")
            print(f"DESCONEXIÓN DEL TAXI {taxi} NO ESPERADA.")
            freePositionMap(taxi)
            taxis.remove(taxi)
            _, customer2Disconnect, _ = manageTaxi(CLEAN) # Limpiar taxis no activos
            sendMessageKafka("Central2Customer", f"CLIENTE {customer2Disconnect} SERVICIO RECHAZADO.")
            mostrar_mapa()

    for customer in customers:
        if customer[0] not in activeCustomers:
            sendMessageKafka("Central2Customer", f"FIN {customer[0]}")
            print(f"DESCONEXIÓN DEL CLIENTE {customer[0]} NO ESPERADA.")
            manageTaxi(DISCONNECT, customer[0]) # Desconectar cliente de taxi
            freePositionMap(customer[0])
            customers.remove(customer)
            mostrar_mapa()

    # Comprobar que los taxis y customers están activos (10 segundos)
def areActives():
    while True:
        activeTaxis = []
        activeCustomers = []

        consumer = kafka.KafkaConsumer("Status", group_id=str(uuid.uuid4()), bootstrap_servers=[f"{KAFKA_IP}:{KAFKA_PORT}"])

        startTime = time.time()
        while True:
            if time.time() - startTime > 11:
                break

            messages = consumer.poll(1000)
            for _, messagesValues in messages.items():
                for msg in messagesValues:
                    msg = msg.value.decode(FORMAT)
                    id = msg.split(" ")[1]

                    try:
                        taxiID = int(id)
                        if taxiID not in activeTaxis:
                            activeTaxis.append(taxiID)

                    except Exception:
                        customerID = id
                        if customerID not in activeCustomers:
                            activeCustomers.append(customerID)

        disableNoActives(activeTaxis, activeCustomers)
        consumer.close()
        time.sleep(1)
```

Comenzaré explicando la segunda función “areActives”. Esta función que no recibe parámetros se mantiene en un bucle infinito con un consumidor leyendo del topic “Status” (con un número random, uuid, como group_id) y cada 10 segundos sale de esta consumición para comprobar si todos los taxis guardados, realmente están activos, y se encuentran dentro del sistema. La segunda función “disableNoActives” con dos parámetros que son los taxis activos y los clientes activos. Esta función verifica si los taxis y clientes guardados en el sistema siguen activos, de ser así se mantienen activos, por lo contrario, se desconectarán. Esta función maneja la desconexión en todos los casos que hemos podido deducir, además se actualizará

el mapa con cada desconexión y se le enviará un mensaje de FIN para que los taxis o clientes que han caído pero se han reconectado sepan que se han caído del sistema también. Estas funciones afectan directamente a las variables globales `taxis` y `customers` qué son las que almacenan los taxis y clientes dentro del sistema. El siguiente módulo o función permite leer todas las solicitudes de los clientes:

```
# CUSTOMERS
# Leer todas las solicitudes de los clientes
def requestCustomers():
    consumer = kafka.KafkaConsumer("Customer2Central", group_id=str(uuid.uuid4()), bootstrap_servers=[f"{KAFKA_IP}:{KAFKA_PORT}"])
    for msg in consumer:
        msg = msg.value.decode(FORMAT)

        if msg == "ACTIVE?":
            sendMessageKafka("Central2Customer", "CENTRAL ACTIVE")
        elif msg.startswith("SOLICITUD DE SERVICIO:"):
            #print(f"Holaaaaaaaaaa: {msg}")
            id = msg.split(" ")[3]
            ubicacion = msg.split(" ")[4]
            destino = msg.split(" ")[5]

            for customer in customers:
                if customer[0] == id:
                    customers.remove(customer)
                    freePositionMap(customer[0])

            isConnected, _, idTaxi = manageTaxi(CONNECT2TAXI, id, ubicacion)
            if isConnected:
                customers.append((id, ubicacion, destino, f"OK. Taxi {idTaxi}"))
                sendMessageKafka("Central2Customer", f"CLIENTE {id} SERVICIO ACEPTADO.")

                locationTuple = getLocation(ubicacion)
                x = locationTuple[0]
                y = locationTuple[1]
                #print(locationTuple)
                sendMessageKafka("Central2Taxi", f"TAXI {idTaxi} DESTINO {x},{y} CLIENTE {id}")

            elif not isConnected:
                customers.append((id, ubicacion, destino, "KO. Servicio denegado."))
                sendMessageKafka("Central2Customer", f"CLIENTE {id} SERVICIO RECHAZADO.")
                freePositionMap(id)

            putCustomerLocation(id, ubicacion)
            mostrar_mapa()
```

Como se puede observar la imagen anterior muestra la función “`requestCustomers`” qué es la que trata con los mensajes recibidos de los clientes, como podría ser la comprobación que la central sigue activa o viva o solicitudes de servicios. Afectando directamente a la variable global `customers` qué es la que se encarga de almacenar en el sistema los clientes activos. Falta por añadir que, con ayuda de otra función (`manageTaxi`), gestiona el asignar un cliente a un taxi.

El siguiente módulo permite enviar mensajes a través de Kafka:

```
# ENVIAR UN MENSAJE A TRAVÉS DE KAFKA
# Crear un productor y enviar un mensaje a través de Kafka con un topic y su mensaje
def sendMessageKafka(topic, msg):
    time.sleep(0.25)
    PRODUCER.send(topic, msg.encode(FORMAT))
    PRODUCER.flush()
```

Este módulo se encarga de enviar mensajes por Kafka proporcionando por parámetros el topic al que debe enviar y el mensaje. El productor llamado “`PRODUCER`” es una variable global que se asigna por primera vez al iniciar el programa (con ciertos valores pasados por parámetros) y no se vuelve a modificar en todo el funcionamiento.

Este módulo abre un socket para aceptar peticiones de autenticación para los taxis:

```
# Abre un socket para aceptar peticiones de autenticación
def connectionSocket(server):
    server.listen()
    print(f"CENTRAL A LA ESCUCHA EN {server}")
    while True:
        conn, addr = server.accept()

        threadAuthTaxi = threading.Thread(target=authTaxi, args=(conn,))
        threadAuthTaxi.start()
```

El funcionamiento de esta función es simple, lo que hace es permanecer en escucha del socket con un cierto puerto y a partir de ahí abre un hilo con cada conexión para poder verificar varios taxis a la vez.

La siguiente función va estrechamente ligada a la anterior, ya que es la que mantiene la funcionalidad de verificar los taxis, con ayuda de “searchTaxiID”:

```
# Se autentica el taxi buscándolo en la base de datos e envía un mensaje
def authTaxi(conn):
    try:
        msgTaxi = conn.recv(HEADER).decode(FORMAT)
        idTaxi = int(msgTaxi.split("#")[1])

        msg2Send = f"VERIFICANDO SOLICITUD DEL TAXI {idTaxi}...\nVERIFICACIÓN NO SUPERADA."
        if searchTaxiID(idTaxi):
            msg2Send = f"VERIFICANDO SOLICITUD DEL TAXI {idTaxi}...\nVERIFICACIÓN SUPERADA."
            if idTaxi not in mapa[0][0]:
                mapa[0][0].insert(0, idTaxi)

        print(msg2Send)
        conn.send(msg2Send.encode(FORMAT))

        mostrar_mapa()

    except ConnectionResetError:
        # print(f"ERROR!! DESCONEXIÓN DEL TAXI {idTaxi} NO ESPERADA.")
        taxis.remove(idTaxi)

    except Exception:
        text = "ID DEL TAXI NO ENCONTRADA EN LA BASE DE DATOS\nVERIFICACIÓN NO SUPERADA."
        print(text)
        conn.send(text.encode(FORMAT))

    finally:
        conn.close()
```

Aquí se verifica que el taxi no se encuentre en la posición inicial del mapa y el hecho de enviar la información de verificación al taxi.

“searchTaxiID” es la que permite a “authTaxi” autenticar con la base de datos un taxi que pretende verificar:

```

# Buscar y activar un taxi por su ID
def searchTaxiID(idTaxi):
    exists = False
    with open("database.txt", "r+") as file:
        modifiedLine = []

    for line in file:
        id = int(line.split(",")[0])

        if idTaxi == id and idTaxi not in taxis: # Si se encuentra esa ID en la base de datos y no es un taxi activo ya
            exists = True
            taxis.append(idTaxi)
            line = line.replace("NO", "OK")

        modifiedLine.append(line)

    file.seek(0)
    file.writelines(modifiedLine)
    file.truncate()

    return exists

```

Su funcionalidad es buscar en la base de datos un taxi que coincida con la ID que solicita verificarse, en el caso que se verifique se activa el taxi en la base de datos y se envía una confirmación de activación al taxi, gracias a “authTaxi”. La base de datos es un fichero de texto.

La siguiente gran función permite recibir incidencias y movimientos de los taxis:

```

# Recibe incidencias y movimientos
def receiveInfoTaxi():
    consumer = kafka.KafkaConsumer("Taxi2Central", bootstrap_servers=[f"{KAFKA_IP}:{KAFKA_PORT}"], group_id=str(uuid.uuid4()))

    beforeStatus = {}
    for msg in consumer:
        msg = msg.value.decode(FORMAT)

        if msg == "ACTIVE?":
            sendMessageKafka("Central2Taxi", "CENTRAL ACTIVE")

        elif msg.split(" ")[2] == "MOVIMIENTO":
            actualizar_mapa_con_movimiento(msg)

        elif msg.split(" ")[2] == "DESTINO":
            taxiID = int(msg.split(" ")[1])
            destinoCoord = msg.split(" ")[3].split(",")
            destino = (int(destinoCoord[0]), int(destinoCoord[1]))
            customerID = msg.split(" ")[5]
            #print(msg)

            if customerID == "CENTRAL":
                manageTaxi(DISCONNECT_COMPLETED, idCustomer="Central")
                mostrar_mapa()

            else:
                destinoTaxi = getLocationByCoords(destino[0], destino[1])
                for customer in customers:
                    if customer[0] == customerID:
                        if destinoTaxi[0] == customer[1]:
                            destinoTuple = getLocation(customer[2])
                            x = destinoTuple[0]
                            y = destinoTuple[1]
                            #print(f"TAXI {taxiID} DESTINO {x},{y} CLIENTE {customerID}")
                            sendMessageKafka("Central2Taxi", f"TAXI {taxiID} DESTINO {x},{y} CLIENTE {customerID}")
                            manageTaxi(CHANGE_DESTINATION, destination=customer[2], idTaxi=taxiID)

                    elif destinoTaxi[0] == customer[2]:
                        manageTaxi(DISCONNECT_COMPLETED, customer[0]) # Desconectar cliente de taxi
                        mostrar_mapa()

            else:
                idTaxi = int(msg.split(" ")[1])
                status = msg.split(" ")[2]

                if idTaxi in beforeStatus and status != beforeStatus[idTaxi]:
                    if beforeStatus[idTaxi] == "KO":
                        manageTaxi(OK, 0, 0, idTaxi)
                    else:
                        manageTaxi(KO, 0, 0, idTaxi)

                mostrar_mapa()

                if status in ["OK", "KO"]:
                    for key in list(beforeStatus.keys()):
                        if key not in taxis:
                            del beforeStatus[key]

                beforeStatus[idTaxi] = status

```

A grandes rasgos, es un consumidor que permite detectar mensajes de los taxis como comprobación de que la central siga activa, de movimientos, de cambios y llegadas a destinos o servicios completados y comprobaciones de incidencias de los taxis.

Esta función permite, gracias a su par de decisiones, obtener todos los datos de un taxi en concreto o de todos:

```

# Obtener todos los datos de un taxi o todos los taxis
ONE = 1 # Obtener solo 1 taxi
ALL = 2 # Obtener todos los taxis activos de la base de datos
def getTaxi(option, idTaxi = 0):
    activeTaxis = []
    with open("database.txt", "r") as file:
        for line in file:
            id = int(line.split(",")[0]) # 1 (ejemplo)
            status = line.split(",")[2] # OK.Parado (ejemplo)
            active = status.split(".")[0] # OK (ejemplo)

            if option == 1 and (id == idTaxi):
                destinationTaxi = line.split(",")[1] # - (ejemplo)
                status = line.split(",")[2] # OK.Parado (ejemplo)
                active = status.split(".")[0] # OK (ejemplo)
                service = status.split("\n")[0] # Parado (ejemplo)
                return (id, destinationTaxi, active, service)

            if option == 2 and (active == "OK" or active == "KO"):
                activeTaxis.append((id, active))

    return activeTaxis

```

De la misma manera que “searchTaxiID” busca en la base de datos toda la información de uno o todos los taxis disponibles, da igual que estén bloqueados por alguna incidencia o no.

La función “manageTaxi” se subdivide en 8 opciones diferentes para poder realizar la gestión de los taxis en la base de datos. Cada opción tiene su diferente función para que se pueda modificar de una mejor manera:

```

def manageTaxi(option, idCustomer = -1, destination = -1, idTaxi = -1):
    modifiedLine = []
    modified = False
    modifiedTaxiID = 0
    customer2Disconnect = 0
    with open("database.txt", "r+") as file:
        for line in file:
            id = int(line.split(",")[0]) # 1 (ejemplo)
            destinationTaxi = line.split(",")[1] # - (ejemplo)
            status = line.split(",")[2] # OK.Parado (ejemplo)
            active = status.split(".")[0] # OK (ejemplo)
            service = status.split(".")[1] # Parado (ejemplo)

            if option == CLEAN:
                line, customer2Disconnect = manageTaxiCLEAN(line, id, service, destinationTaxi, status)

            elif option == CONNECT2TAXI or option == CONNECT2TAXI_ID:
                line, modified, modifiedTaxiID = manageTaxiCONNECT2TAXI(line, option, id, idTaxi, destinationTaxi, active, destination, idCustomer, modified, modifiedTaxiID)

            elif option == DISCONNECT or option == DISCONNECT_COMPLETED:
                line = manageTaxiDISCONNECT(line, option, id, service, idCustomer, destinationTaxi, status)

            elif option == CHANGE_DESTINATION:
                line = manageTaxiCHANGE_DESTINATION(line, id, idTaxi, destinationTaxi, destination)

            elif option == OK:
                line = manageTaxiOK(line, id, idTaxi, active)

            elif option == KO:
                line = manageTaxiKO(line, id, idTaxi, active)

            modifiedLine.append(line)

        file.seek(0)
        file.writelines(modifiedLine)
        file.truncate()

    return modified, customer2Disconnect, modifiedTaxiID

```

Cada función podrá leer y modificar la base de datos de diferentes maneras.

La primera opción “CLEAN” permite deshabilitar los taxis no activos y desconectar al posible cliente asociado.

```
# Gestionar los taxis
CLEAN = 1 # Deshabilitar los no activos (y desconecta el cliente que podría tener asociado)
def manageTaxiCLEAN(line, id, service, destinationTaxi, status):
    customer2Disconnect = 0
    if id not in taxis:
        if service.startswith("Servicio"):
            customer2Disconnect = (service.split(" ")[1]).split("\n")[0]

        line = line.replace(destinationTaxi, "-")
        line = line.replace(status, "NO.Parado\n")

    return line, customer2Disconnect
```

Todo ello modificando la base de datos, fichero.

Las siguientes opciones son “CONNECT2TAXI” y “CONNECT2TAXI_ID”:

```
CONNECT2TAXI = 2 # Añadir un servicio (cliente) a un taxi
CONNECT2TAXI_ID = 2.1 # Añadir un servicio (central) a un taxi en concreto
def manageTaxiCONNECT2TAXI(line, option, id, idTaxi, destinationTaxi, active, destination, idCustomer, modified, modifiedTaxiID):
    if CONNECT2TAXI == option and not modified and destinationTaxi == "-" and active == "OK":
        for location in locations:
            if location[0] == destination:
                modified = True
                modifiedTaxiID = id
                line = line.replace("-", destination)
                line = line.replace("Parado", f"Servicio {idCustomer}")
                break

    elif CONNECT2TAXI_ID == option and idTaxi == id and destinationTaxi == "-" and active == "OK":
        if idCustomer == "Central":
            modified = True
            modifiedTaxiID = id
            line = line.replace("-", destination)
            line = line.replace("Parado", f"Servicio {idCustomer}")

    return line, modified, modifiedTaxiID
```

Esta opción dividida en dos permite conectar un cliente al primer taxi disponible en la base de datos o a un taxi en concreto.

A continuación se muestra otra opción con dos subopciones:

```
DISCONNECT = 3 # Liberar un taxi de su servicio (cliente) debido a un error
DISCONNECT_COMPLETED = 3.1 # Liberar un taxi de su servicio (cliente) por servicio completado
def manageTaxiDISCONNECT(line, option, id, service, idCustomer, destinationTaxi, status):
    if service == f"Servicio {idCustomer}\n":
        line = line.replace(destinationTaxi, "-")
        line = line.replace(status, "OK.Parado\n")

    if DISCONNECT == option:
        sendMessageKafka("Central2Taxi", f"TAXI {id} ERROR CUSTOMER")
    elif DISCONNECT_COMPLETED == option:
        sendMessageKafka("Central2Customer", f"CLIENTE {idCustomer} SERVICIO COMPLETADO.")

    return line
```

La opción de “DISCONNECT” permite liberar un taxi de su servicio debido a un error y “DISCONNECT_COMPLETED” permite liberar un taxi de su cliente a causa de un servicio completado.

Y finalmente las tres últimas opciones:

```

CHANGE_DESTINATION = 4 # Cambiar el destino de un taxi
def manageTaxiCHANGE_DESTINATION(line, id, idTaxi, destinationTaxi, destination):
    if id == idTaxi:
        #print(idTaxi)
        line = line.replace(destinationTaxi, destination)

    return line

OK = 5 # OK en taxi
def manageTaxiOK(line, id, idTaxi, active):
    if id == idTaxi and active == "KO":
        line = line.replace(active, "OK")

    return line

KO = 6 # KO en taxi
def manageTaxiKO(line, id, idTaxi, active):
    if id == idTaxi and active == "OK":
        line = line.replace(active, "KO")

    return line

```

“CHANGE_DESTINATION” es la opción que permite cambiar el destino de un taxi en concreto por otro destino pasado por parámetro. “OK” y “KO” cambia el estado del taxi por cualquier razón. Todas las opciones de la función padre “manageTaxi” trabajan sobre la base de datos.

La siguiente función permite buscar a un cliente por su ID:

```

# Buscar un CUSTOMER según su ID
def searchCustomerID(idCustomer):
    for customer in customers:
        if customer[0] == idCustomer:
            return customer

    return -1

```

Lo realiza accediendo a la variable global `customers`.

La función “actualizar_mapa_con_movimiento” la explicaré en dos partes:

```
# Función encargada de actualizar el mapa
def actualizar_mapa_con_movimiento(mensaje):
    global mapa
    words = mensaje.split(" ")
    #print(mensaje)
    idTaxi = int(words[1])
    direccion = words[3]
    x = int(words[5])
    y = int(words[6])
```

Esta es la primera parte de la función y se encarga de dividir el mensaje que recibe del taxi, en diferentes variables para su posterior interpretación. Además se puede observar que accederá a una variable global llamada “mapa” que almacena todas las posiciones del mapa.

Y la imagen de la parte de abajo es la segunda parte:

```

# Ajustar las coordenadas de 1-20 a 0-19 para usar en la matriz `mapa`
x -= 1
y -= 1

serviceCustomerID = -1
# Limpiar la posición anterior del taxi
for row in range(len(mapa)):
    for col in range(len(mapa[row])):
        try:
            taxiIndex = mapa[row][col].index(idTaxi)
        except:
            taxiIndex = -1

        if taxiIndex != -1 and isinstance(mapa[row][col][taxiIndex], int) and mapa[row][col][taxiIndex] == idTaxi:
            if len(mapa[row][col]) > 1:
                #print(mapa[row][col][0])
                #print(idTaxi)
                #print(taxiIndex)
                mapa[row][col].pop(taxiIndex) # Cambia la posición anterior a posición vacía

            taxi = getTaxi(ONE, idTaxi)
            destinationTaxi = taxi[1]
            activeTaxi = taxi[2]
            serviceTaxi = taxi[3]

            if activeTaxi == "OK" and serviceTaxi.startswith("Servicio"):
                idCustomer = serviceTaxi.split(" ")[1]

                customer = searchCustomerID(idCustomer)

                # Si el destino del TAXI coincide con el destino del cliente que esta en esa casilla se ponen juntas
                if idCustomer in mapa[row][col] and destinationTaxi == customer[2] and len(mapa[row][col]) > 1:
                    #print(mapa[row][col])
                    customerIndex = mapa[row][col].index(customer[0])
                    serviceCustomerID = mapa[row][col][customerIndex]
                    mapa[row][col].pop(customerIndex) # Cambia la posición anterior a posición vacía

                    if customer[3] != "OK":
                        i = 0
                        for customer in customers:
                            if customer[0] == idCustomer:
                                customers[i] = (customer[0], customer[1], customer[2], "OK")

                        i += 1

# Actualiza la nueva posición en el mapa según la dirección
if 0 <= x < len(mapa) and 0 <= y < len(mapa[0]):
    if dirección in ["Norte", "Sur", "Este", "Oeste", "Noreste", "Sureste", "Noroeste", "Suroeste"]:
        # Guarda el número del taxi en el mapa para ser coloreado en la función mostrar_mapa
        if serviceCustomerID != -1 and serviceCustomerID not in mapa[x][y]:
            mapa[x][y].insert(0, serviceCustomerID)

        if idTaxi not in mapa[x][y]:
            mapa[x][y].insert(0, idTaxi)

    else:
        print(f"[ERROR] Dirección no reconocida: {dirección}")

# Muestra el mapa actualizado
mostrar_mapa()

```

En esta se puede observar que se accede a todas las posiciones del mapa que cada una de ellas son una lista, por lo que se pueden almacenar varias posiciones en ella. A grandes rasgos, lo que realiza esta función es mover un taxi en el mapa, en el caso que lleve a un cliente éste lo moverá con él. Primero borra la posición o posiciones (en caso de errores) en las que se encuentran el taxi y/o el cliente y, posteriormente, los inserta en la nueva posición y actualiza el mapa.

Este módulo permite liberar una posición del mapa a partir de la id pasada como parámetro:

```
def freePositionMap(id):
    for row in range(len(mapa)):
        for col in range(len(mapa[row])):
            try:
                idIndex = mapa[row][col].index(id)
            except:
                idIndex = -1

            if idIndex != -1 and mapa[row][col][idIndex] == id:
                if len(mapa[row][col]) > 1:
                    mapa[row][col].pop(idIndex) # Cambia la posición anterior a posición vacía
```

Como tiene que liberar una posición del mapa debe acceder a la variable global “mapa”.

La función “putCustomerLocation” permite como dice su nombre poner un cliente en una localización en el mapa:

```
# Poner a un consumidor en una localización
def putCustomerLocation(customerID, locationID):
    for location in locations:
        if location[0] == locationID:
            locationPosX = location[1]
            locationPosY = location[2]

            if customerID not in mapa[locationPosX-1][locationPosY-1]:
                mapa[locationPosX-1][locationPosY-1].insert(0, customerID)

    return True

return False
```

Se realiza utilizando las localizaciones obtenidas a partir del fichero de localizaciones.

```

# Localizaciones de EC_locations.json
def loadLocations():
    global mapa

    try:
        with open("EC_locations.json", "r", encoding="utf-8") as locats:
            x = json.load(locats)

            for location in x["locations"]:
                locationID = location["Id"]
                locationPos = location["POS"]
                locationPosX = int(locationPos.split(",")[0])
                locationPosY = int(locationPos.split(",")[1])

                locations.append((locationID, locationPosX, locationPosY))

                mapa[locationPosX-1][locationPosY-1].pop(0)
                mapa[locationPosX-1][locationPosY-1].insert(0, locationID)

    except Exception:
        print(F"NO EXISTE EL ARCHIVO (EC_locations.json)")

```

Esta función es la que abre el archivo de localizaciones y los escribe en el mapa, además de guardarlos en una variable global llamada “locations”, la cuál guarda su ID la localización en x y en y.

Las siguientes dos funciones “getLocationByCoords” y “getLocation” tienen la misma funcionalidad:

```

# Obtener una localización según las coordenadas
def getLocationByCoords(x, y):
    for location in locations:
        if location[1] == x and location[2] == y:
            return location

    return None

# Obtener una localización según la ID de una localización
def getLocation(customerLocation):
    for location in locations:
        if location[0] == customerLocation:
            return (location[1], location[2])

```

Las dos sirven para obtener la localización completa (ID, x, y) o al menos las variables x e y según unas coordenadas o según su ID.

```

# Mostrar taxis, localizaciones, clientes y movimientos en el mapa
def showMapObjects(mapa2Print):
    for row in range(len(mapa)):
        # Mostrar el número de fila (ajustado de 1-20)
        mapa2Print.append(f"{row + 1:2} ")
        for col in range(len(mapa[row])):
            mapValue = mapa[row][col][0]

            #print(" ", end="")
            #print(mapa[row][col], end=" ")

            if True:
                if isinstance(mapValue, int): # Taxis
                    taxi = getTaxi(ONE, mapValue)
                    destinationTaxi = taxi[1]
                    activeTaxi = taxi[2]
                    serviceTaxi = taxi[3]

                    if activeTaxi == "OK" and serviceTaxi.startswith("Servicio"):
                        idCustomer = serviceTaxi.split(" ")[1]

                        customer = searchCustomerID(idCustomer)

                        # Si el destino del TAXI coincide con el destino del cliente que esta en esa casilla
                        if idCustomer in mapa[row][col] and destinationTaxi == customer[2]:
                            mapa2Print.append(Fore.GREEN + f"{mapValue:2}{idCustomer}" + Style.RESET_ALL)
                        else:
                            mapa2Print.append(Fore.GREEN + f"{mapValue:2} " + Style.RESET_ALL)

                    elif activeTaxi == "OK" and serviceTaxi == "Parado":
                        mapa2Print.append(Fore.RED + f"{mapValue:2} " + Style.RESET_ALL)

                    elif activeTaxi == "KO":
                        mapa2Print.append(Fore.RED + f"{mapValue:2}!" + Style.RESET_ALL)

                    elif isinstance(mapValue, str) and len(mapValue) == 1 and mapValue.isalpha():
                        if mapValue.isupper(): # Localizaciones
                            mapa2Print.append(Fore.BLUE + f" {mapValue} " + Style.RESET_ALL)
                        else: # Clientes
                            mapa2Print.append(Fore.YELLOW + f" {mapValue} " + Style.RESET_ALL)

                    else:
                        mapa2Print.append(f" {mapValue} ")

                mapa2Print.append("\n")

```

La función “showMapObjects” recibe como parámetro el string ya que es un fragmentación de la función “mostrar_mapa” que permite obtener todo el mapa junto a la cuadrícula de acciones en un string el cual se mostraría tanto en la central como en los clientes y taxis. En concreto esta función halla los colores y el formato que debe mostrar todos los taxis, clientes y localizaciones de todo el mapa.

```

def mostrar_mapa():
    mapa2Print.clear()

    mapa2Print.append("\n" * 6)

    mapa2Print.append(LINE)
    mapa2Print.append(f"{' ':<15} *** EASY CAB Release 1 ***\n")
    mapa2Print.append(LINE)

    # Mostrar encabezado de taxis y clientes
    mapa2Print.append(f"{' ':<10} {'Taxis':<19} {'|':<8} {'Clientes'}\n")
    mapa2Print.append(LINE)
    mapa2Print.append(f"{' ':<3} {'Id.':<5} {'Destino':<10} {'Estado':<9} {'|':<2} {'Id.':<5} {'Destino':<10} {'Estado':<10}\n")
    mapa2Print.append(LINE)

    # Mostrar los taxis y los clientes en filas paralelas
    maxSize = max(len(taxis), len(customers))
    for i in range(maxSize):
        taxi_line = f"{SPACE * 3}"
        cliente_line = f"{SPACE * 3}"

        if i < len(taxis):
            taxi = getTaxi(ONE, taxis[i])
            if taxi:
                idTaxi = taxi[0]
                destTaxi = taxi[1]
                activeTaxi = taxi[2]
                serviceTaxi = taxi[3].split("\n")[0]

                stateTaxi = ""
                if serviceTaxi.startswith("Servicio"):
                    stateTaxi = f"{activeTaxi}. {serviceTaxi}"
                else:
                    stateTaxi = f"{SPACE * 3} {activeTaxi}. {serviceTaxi}"

                taxi_line = f"{SPACE*4} {idTaxi} {SPACE*5} {destTaxi} {stateTaxi} "
                if activeTaxi == "KO":
                    taxi_line = Fore.RED + taxi_line + Style.RESET_ALL
                taxi_line += "|"

            else:
                taxi_line = SPACE * 30

        if i < len(customers):
            customer = customers[i]
            idCustomer = customer[0]
            ubicustomer = customer[1]
            destCustomer = customer[2]
            stateCustomer = customer[3]

            cliente_line = f"{idCustomer:<5} {destCustomer:<10} {stateCustomer}"

        mapa2Print.append(f"{taxi_line} {cliente_line}\n")

        if i == maxSize - 1:
            mapa2Print.append(LINE)

    mapa2Print.append(LINE)
    mapa2Print.append(" " + ".join([f"{i:2}" for i in range(1, 21)]) + "\n")
    mapa2Print.append(LINE)

    showMapObjects(mapa2Print)

    mapa2Print.append(LINE)
    mapa2Print.append(f"{' ':<16} Estado general del sistema: OK\n")
    mapa2Print.append(LINE)

```

Esta función gestiona la lógica de la cuadrícula de estados y los añade a una lista que posteriormente se mostrará por pantalla y se enviará por Kafka para que en el cliente y el taxi también se muestre.

```

fullMapa2Print = "" .join(mapa2Print)
print(fullMapa2Print)

time.sleep(0.25)
sendMessageKafka("Mapa", fullMapa2Print)

```

La función “mostrar_mapa” se encarga de mostrar el mapa por pantalla, es decir, por consola, siguiendo el formato establecido en la práctica, quedándose de esta manera (un ejemplo):

*** EASY CAB Release 1 ***																					
Taxis						Clientes															
Id.		Destino		Estado		Id.		Destino		Estado											
4		a		OK. Servicio a		a		A		OK. Taxi 4											
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20		
1	4		
2		
3		
4		
5	.	.	A		
6		
7		
8	C		
9		
10		
11	F		
12		
13		
14		
15		
16		
17	D	.	.	.		
18		
19	E		
20		
Estado general del sistema: OK																					

Digital Engine -> EC_DE

Aplicación encargada de manejar la lógica principal de todo el sistema distribuido que involucra a un taxi autónomo.

Esta aplicación incluye los siguientes módulos:

<heuristica(pos_actual, pos_destino)>

```
# Para el algoritmo A* --> heurística --> distancia euclídea
def heuristica(pos_actual, pos_destino):
    return math.sqrt((pos_actual[0] - pos_destino[0]) ** 2 + (pos_actual[1] - pos_destino[1]) ** 2)
```

Recibe la posición actual y la de destino por parámetros, para calcular la distancia euclídea, utilizando como heurística para el algoritmo A* con movimiento esférico.

<movimiento_esferico(posicion, movimiento)>

```
# Para el algoritmo A* --> movimiento esférico
def movimiento_esferico(posicion, movimiento):
    nueva_posX = (posicion[0] + movimiento[0] - 1) % TAM_MAPA + 1
    nueva_posY = (posicion[1] + movimiento[1] - 1) % TAM_MAPA + 1
    return (nueva_posX, nueva_posY)
```

Recibe por parámetros la posición y el movimiento, ajustando las coordenadas del mapa para simular un movimiento esférico, es decir, que tenga los bordes conectados.

<algoritmo_a_estrella(inicio, destino)>

```
# Algoritmo A*
def algoritmo_a_estrella(inicio, destino):
    listaFrontera = []
    heapq.heappush(listaFrontera, (0,inicio))

    listaInterior = set()

    coste = {inicio: 0}
    rastreo = {inicio: None}

    while listaFrontera:
        _, posicion_actual = heapq.heappop(listaFrontera)

        if posicion_actual == destino:
            camino = []

            while posicion_actual is not None:
                camino.append(posicion_actual)
                posicion_actual = rastreo[posicion_actual]

            return camino[::-1]

        listaInterior.add(posicion_actual)

        for movimiento in MOVIMIENTOS:
            vecino = movimiento_esferico(posicion_actual, movimiento)

            if vecino in listaInterior:
                continue

            nuevo_coste = coste[posicion_actual] + 1

            if vecino not in coste or nuevo_coste < coste[vecino]:
                coste[vecino] = nuevo_coste

                prioridad = nuevo_coste + heuristica(vecino, destino)

                heapq.heappush(listaFrontera, (prioridad, vecino))

                rastreo[vecino] = posicion_actual

    return None
```

Recibe como argumentos el inicio y el destino, y es donde se implementa el algoritmo A* para encontrar el camino más corto entre el inicio y el destino.

<mover_taxi(destino, customer=None)>

```
# Función encargada de mover el taxi y enviar la nueva posición a Kafka
camino_pendiente = None # Variable global para almacenar el camino pendiente

def mover_taxi(destino, customer=None):
    global posicion_actual, camino_pendiente, customerERROR

    end = False
    while not disconnect and not end and not customerERROR: # Bucle para permitir que el taxi revise su estado continuamente
        # Si hay un camino pendiente desde una pausa anterior (estado KO), lo reutilizamos
        if camino_pendiente:
            camino = camino_pendiente
            camino_pendiente = None # Limpiamos el camino pendiente ya que vamos a continuar
        else:
            # Si no hay camino pendiente, calculamos uno nuevo
            camino = algoritmo_a_estrella((posicion_actual['x'], posicion_actual['y']), destino)

        if camino is None:
            print(f"[ERROR] No se pudo encontrar un camino al destino {destino}")
            return

        paso_anterior = (posicion_actual['x'], posicion_actual['y'])

        camino_pendiente, end = moveStepTaxi(camino, destino, paso_anterior, camino_pendiente, customer)

        # Aquí puedes agregar un pequeño retraso antes de volver a revisar el estado
        time.sleep(1) # Espera 1 segundo antes de volver a verificar el estado

    if customerERROR:
        customerERROR = False
```

Recibe como parámetros un destino y un cliente, y mueve el taxi hacia el destino usando el algoritmo A*, y reanuda el camino tras pausas por el estado “KO”.

<moveStepTaxi(camino, destino, paso_anterior, camino_pendiente, custommer)>

```
# Mover un paso el taxi
def moveStepTaxi(camino, destino, paso_anterior, camino_pendiente, customer):
    global beforeTaxiState
    # Mueve el taxi por cada paso en el camino
    for paso in camino:
        nueva_posX, nueva_posY = paso

        # Verificamos si el estado es KO
        if beforeTaxiState[0] == "KO":
            #print(f"[KO] Taxi {ID_TAXI} está en KO, guardando el camino pendiente.")
            camino_pendiente = camino[camino.index(paso):] # Guardamos el camino restante
            break # Salimos del bucle for, pero no del bucle while

        if not disconnect and not customerERROR:
            dirección = determinar_dirección(paso_anterior, (nueva_posX, nueva_posY))

            posicion_actual['x'] = nueva_posX
            posicion_actual['y'] = nueva_posY

            #print(f"[MOVIMIENTO] Taxi {id_taxi} se ha movido a la posición: {posicion_actual}")

            # Enviar la nueva posición y dirección a Kafka
            if posicion_actual is not None:
                sendMessageKafka("Taxi2Central", f"TAXI {ID_TAXI} MOVIMIENTO {dirección} POSICIÓN {posicion_actual['x']} {posicion_actual['y']}")

            else:
                print(f"ERROR!! TAXI NO TIENE UNA POSICIÓN DEFINIDA PARA ENVIAR")

            # Verificamos si el taxi ha llegado a su destino
            if (nueva_posX, nueva_posY) == destino:
                if customer != None:
                    sendMessageKafka("Taxi2Central", f"TAXI {ID_TAXI} DESTINO {destino[0]}, {destino[1]} CLIENTE {customer}")

                return camino_pendiente, True # Salimos de la función, no se enviarán más movimientos

            # Actualizar el paso anterior
            paso_anterior = (nueva_posX, nueva_posY)

        time.sleep(3) # Simula el tiempo de movimiento

    return camino_pendiente, False
```

Recibe como parámetros el camino, el destino, el paso anterior, el camino pendiente y el cliente, de manera que realiza un paso en el camino del taxi, envía la posición actual a Kafka, y verifica si llegó al destino.

<determinar_direccion(origen, destino)>

```
# Determina la dirección del movimiento basado en las coordenadas de origen y destino
def determinar_direccion(origen, destino):
    dx = destino[0] - origen[0]
    dy = destino[1] - origen[1]

    if dx == 0 and dy > 0:
        return "Norte"
    elif dx == 0 and dy < 0:
        return "Sur"
    elif dx > 0 and dy == 0:
        return "Este"
    elif dx < 0 and dy == 0:
        return "Oeste"
    elif dx > 0 and dy > 0:
        return "Noreste"
    elif dx > 0 and dy < 0:
        return "Sureste"
    elif dx < 0 and dy > 0:
        return "Norooeste"
    else:
        return "Suroeste"
```

Recibe como parámetros un origen y un destino, determinando la dirección del movimiento entre esas dos posiciones.

<conexion_central(ip_central, port_central)>

```
# Función encargada de manejar la conexión con la central y esperar órdenes, usando sockets
def conexion_central(ip_central, port_central):
    try:
        # Creación de un socket IP/TCP
        # AF_INET -> se usará IPv4
        # SOCK_STREAM -> se usará TCP
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as socket_creado:
            # Conexión del socket creado con el servidor (central de control) mediante la IP y PUERTO dados
            socket_creado.connect((ip_central, int(port_central)))

            # Autenticación con la central enviando el ID del taxi
            mensaje_de_autenticacion = f'AUTENTICAR TAXI #{ID_TAXI}'

            # Envío del mensaje de autenticación
            socket_creado.send(mensaje_de_autenticacion.encode(FORMAT))

            # Espera de la respuesta de la central
            respuesta = socket_creado.recv(HEADER).decode(FORMAT)

            # Mostramos la respuesta de la central
            print(f'{respuesta}')

            # Comprobamos si la autenticación fue válida o errónea
            if respuesta.split("\n")[1] == "VERIFICACIÓN SUPERADA.":
                sendMensajeKafka("Status", f"TAXI {ID_TAXI} ACTIVO.")
                return socket_creado # Devolvemos el socket para usarlo después
            else:
                return None

    except ConnectionError:
        print(f'Error de conexión con la central en {ip_central}:{port_central}')

    except Exception as e:
        print(f'Error generado: {str(e)}')
        return None
```

Recibe como parámetros la IP y el puerto de la central, realizando la autenticación del taxi con la central a través de un socket.

<sendMessageKafka(topic, msg)>

```
# Crear un productor y enviar un mensaje a través de Kafka con un topic y su mensaje
def sendMessageKafka(topic, msg):
    time.sleep(0.2)
    PRODUCER.send(topic, msg.encode(FORMAT))
    PRODUCER.flush()
```

Recibe como argumentos el topic y el mensaje que se va a enviar por Kafka a la central por el topic introducido como argumento.

<receiveServices()>

```
# Recibe un servicio, una comprobación o un mensaje final
def receiveServices():
    global customerERROR, disconnect

    consumer = kafka.KafkaConsumer("Central2Taxi", group_id=str(uuid.uuid4()), bootstrap_servers=f"{KAFKA_IP}:{KAFKA_PORT}")

    while not disconnect:
        messages = consumer.poll(5000)
        for _, messagesValues in messages.items():
            for msg in messagesValues:
                msg = msg.value.decode(FORMAT)
                #print(msg)

                if msg.startswith(f"TAXI {ID_TAXI},CENTRAL "):
                    taxiAction = msg.split(",")[-1].split(" ")[1]

                    if taxiAction.startswith("DESTINO"):
                        destino = taxiAction.split(" ")[1]

                        threadMoverTaxi = threading.Thread(target=mover_taxi, args=(destino, "central"))
                        threadMoverTaxi.start()

                    elif taxiAction == "BASE":
                        destino = (1, 1)

                        threadMoverTaxi = threading.Thread(target=mover_taxi, args=(destino, "base"))
                        threadMoverTaxi.start()

                    elif taxiAction == "PARAR":
                        incidenciasControl("KO")

                    elif taxiAction == "REANUDAR":
                        incidenciasControl("OK")

                elif msg.startswith(f"TAXI {ID_TAXI} DESTINO"):
                    destinoCoord = msg.split(" ")[3].split(",")
                    destino = (int(destinoCoord[0]), int(destinoCoord[1]))
                    #print(destino)
                    customer = msg.split(" ")[5]

                    threadMoverTaxi = threading.Thread(target=mover_taxi, args=(destino, customer))
                    threadMoverTaxi.start()

                elif msg == f"TAXI {ID_TAXI} ERROR CUSTOMER":
                    customerERROR = True

                elif msg == f"FIN {ID_TAXI}":
                    disconnect = True
                    consumer.close()
```

No recibe argumentos, pero recibe a través de Kafka mensajes de la central, como son destino, base, parar y reanudar, para que el taxi realice estas acciones.

<enviar_estado_a_central(estado, incidencia=None)>

```
# Función encargada de enviar la incidencia a la central usando kafka
def enviar_estado_a_central(estado, incidencia=None):
    if incidencia:
        mensaje = f"TAXI {ID_TAXI} {estado} INCIDENCIA {incidencia}"
    else:
        mensaje = f"TAXI {ID_TAXI} {estado}"

    sendMessageKafka("Taxi2Central", mensaje)
    #print(mensaje)

    #print(f"[KAFKA] Estado enviado a la central: {mensaje}")
```

Recibe por parámetros el estado y la incidencia, y envía esta información a la central a través de Kafka.

<incidenciasControl(estado, incidencia=None, incidencia_detectada="", sensor=False)>

```
def incidenciasControl(estado, incidencia=None, incidencia_detectada="", sensor=False):
    global beforeTaxiState
    modified = False

    if beforeTaxiState[0] == "OK" and estado == "KO":
        #print("[CUIDADO] Incidencia recibida")
        beforeTaxiState[0] = "KO"
        beforeTaxiState[1] = sensor
        incidencia_detectada = incidencia
        modified = True
    elif beforeTaxiState[0] == "KO" and beforeTaxiState[1] == True and estado == "OK" and sensor == True:
        #print("[SOLUCIONADO] Incidencia solucionada")
        beforeTaxiState[0] = "OK"
        beforeTaxiState[1] = True
        modified = True
    elif beforeTaxiState[0] == "KO" and beforeTaxiState[1] == False and estado == "OK" and sensor == False:
        #print("[SOLUCIONADO] Incidencia solucionada")
        beforeTaxiState[0] = "OK"
        beforeTaxiState[1] = False
        modified = True

    #beforeTaxiState = estado

    if modified:
        | enviar_estado_a_central(estado, incidencia)
    #print(f"[ESTADO] Estado recibido: {estado}")

    return incidencia_detectada
```

Recibe como argumentos el estado, la incidencia, la incidencia_detectada y el sensor como variable booleana, y se encarga de gestionar y actualizar el estado del taxi, OK/KO, en función de incidencias recibidas de los sensores.

<recibir_estado_sensor(ip_sensores, port_sensores)>

```
# Función encargada de recibir el estado de los sensores
def recibir_estado_sensor(ip_sensores, port_sensores):
    global disconnect, sensorActivity
    addr_DE = (ip_sensores, int(port_sensores))

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as servidor:
        servidor.bind(addr_DE)
        servidor.listen()
        print(f"[ESCUCHA] Esperando estado de sensores en {ip_sensores}:{port_sensores}...")

        incidencia_detectada = ""

        while not disconnect:
            conn, _ = servidor.accept()
            with conn:
                while not disconnect:
                    try:
                        estado = conn.recv(HEADER).decode(FORMAT)

                        if "INCIDENCIA" in estado:
                            estado, incidencia = estado.split(", INCIDENCIA: ")
                        else:
                            incidencia = None

                        incidencia_detectada = incidenciasControl(estado, incidencia, incidencia_detectada, sensor=True)

                    except Exception as e:
                        print("Sensor caído")
                        print("Taxi sin sensor, peligro de accidente, taxi parado")
                        disconnect = True
```

Recibe como argumentos la dirección IP u el puerto de los sensores, para escuchar el estado de los sensores y actualizar el estado llamando a incidenciasControl.

<esperarConexionSensor(ip_sensores, port_sensores)>

```
# Función encargada de recibir la conexión del sensor con el digital engine
def esperarConexionSensor(ip_sensores, port_sensores):
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as servidor:
            servidor.bind((ip_sensores, int(port_sensores)))
            servidor.listen()
            print(f"[ESPERA] Esperando conexión del sensor en {ip_sensores}:{port_sensores}...")

            conn, addr = servidor.accept()
            print(f"[CONECTADO] Sensor conectado desde {addr}")
            return conn
    except Exception as e:
        print(f"[ERROR] No se pudo establecer conexión con el sensor: {str(e)}")
    return None
```

Recibe como argumentos la dirección IP y el puerto de los sensores, para esperar la conexión inicial del sensor con el taxi, y verifique la disponibilidad de este.

<showMap()>

```
# Mostrar mapa
def showMap():
    global disconnect

    consumer = kafka.KafkaConsumer("Mapa", group_id=str(uuid.uuid4()), bootstrap_servers=f"{KAFKA_IP}:{KAFKA_PORT}")

    while not disconnect:
        messages = consumer.poll(1000)
        for _, messagesValues in messages.items():
            for msg in messagesValues:
                msg = msg.value.decode(FORMAT)
                print(msg, end="")
```

No recibe nada como parámetros, y muestra el estado actual del mapa en la consola recibiendo datos en tiempo real desde Kafka.

<sendCentralActive()>

```
# Envía cada 2 segundos un mensaje a la central para comprobar su actividad
def sendCentralActive():
    while not disconnect:
        sendMessageKafka("Taxi2Central", f"ACTIVE?")
        sendMessageKafka("Status", f"TAXI {ID_TAXI} ACTIVO.")
        time.sleep(2)
```

No recibe nada como parámetros, y envía mensajes de verificación de actividad a la central cada dos segundos.

<statusControl()>

```

# Recibe el STATUS de la CENTRAL
def statusControl():
    global disconnect
    consumer = KafkaConsumer("Central2Taxi", group_id=str(uuid.uuid4()), bootstrap_servers=f"{KAFKA_IP}:{KAFKA_PORT}")

    startTime = time.time()
    while not disconnect:
        endTime = time.time()
        if endTime - startTime > 10:
            print("ERROR!! SE HA PERDIDO LA CONEXIÓN CON LA CENTRAL.")
            disconnect = True
            break

        messages = consumer.poll(1000)
        for _, messagesValues in messages.items():
            for msg in messagesValues:
                msg = msg.value.decode(FORMAT)
                if msg == "CENTRAL_ACTIVE":
                    startTime = time.time()

    consumer.close()

```

No recibe nada como parámetros, y controla el estado de la conexión con la central, desconectando el taxi en el caso en que se pierda la conexión.

<main()>

```

def main(ip_central, port_central, ip_sensores, port_sensores):
    # Conexión con el sensor
    conexion_recibida_sensor = esperarConexionSensor(ip_sensores, port_sensores)

    if conexion_recibida_sensor:
        # Conexión con la central
        conexion_verificada = conexionCentral(ip_central, port_central)

        if conexion_verificada:
            threadRecibeEstados = threading.Thread(target=recibirEstadoSensor, args=(ip_sensores, port_sensores))
            threadRecibeEstados.start()

            threadServices = threading.Thread(target=receiveServices)
            threadServices.start()

            threadMapa = threading.Thread(target=showMap)
            threadMapa.start()

            threadActiveCentralMSG = threading.Thread(target=sendCentralActive)
            threadActiveCentralMSG.start()

            threadStatusControlCentral = threading.Thread(target=statusControl)
            threadStatusControlCentral.start()

        else:
            print(f"Falló la autenticación, taxi {ID_TAXI} inhabilitado, no se encuentra en la base de datos")
            sys.exit(1)

    else:
        print(f"Falló la conexión con el sensor")
        sys.exit(1)

```

Recibe como argumentos la dirección IP y el puerto tanto de la central como de los sensores, y configura y lanza las conexiones con el sensor y la central, iniciando los hilos de cada funcionalidad.

bloque común en python, encargado de ejecutar el script principal y manejar los parámetros desde consola, es decir, desde la línea de comandos

```

# Main
if __name__ == "__main__":
    # Se esperan 8 argumentos
    if len(sys.argv) == 8:
        ip_central = sys.argv[1]
        port_central = sys.argv[2]
        KAFKA_IP = sys.argv[3]
        KAFKA_PORT = sys.argv[4]
        ip_sensores = sys.argv[5]
        port_sensores = sys.argv[6]
        ID_TAXI = sys.argv[7]

        PRODUCER = kafka.KafkaProducer(bootstrap_servers=f"{KAFKA_IP}:{KAFKA_PORT}")

        # Conexión del taxi con la central y kafka
        main(ip_central, port_central, ip_sensores, port_sensores)

    else:
        print("ERROR!! Falta por poner <IP EC_CENTRAL> <PUERTO EC_CENTRAL> <IP BOOTSTRAP-SERVER> <PUERTO BOOTSTRAP-SERVER> <IP EC_S> <PUERTO EC_S> <ID TAXI>")

```

Esta aplicación utiliza las siguientes librerías y las siguientes variables globales:

```
# Librerías
import socket # Para establecer la conexión de red entre las aplicaciones del sistema
import sys # Para acceder a los argumentos de la línea de comandos
import kafka
import time
import threading
import uuid

# Para el algoritmo A* --> librerías
import heapq
import math

HEADER = 64
FORMAT = 'utf-8'
FIN = 'FIN'
KAFKA_IP = 0
KAFKA_PORT = 0
PRODUCER = 0
ID_TAXI = 0

# Inicialización de la posición del taxi (indefinida)
posición_actual = {'x': 1, 'y': 1}

# Para el algoritmo A* --> dimensiones del mapa
TAM_MAPA = 20

# Para el algoritmo A* --> movimientos
MOVIMIENTOS = [(0,1), (0,-1), (1,0), (-1,0), (1,1), (1,-1), (-1,1), (-1,-1)]
beforeTaxiState = ["OK", False] # Estado del taxi y si ese estado proviene del sensor
sensorActivity = False
customerERROR = False
disconnect = False
```

HEADER = 64; Define el tamaño del encabezado en bytes para los mensajes que se envían a través del socket.

FORMAT = 'utf-8'; Define el formato de codificación de caracteres para enviar y recibir mensajes de texto en el socket.

FIN = 'FIN'; Representa un mensaje de finalización.

KAFKA_IP = 0 Y KAFKA_PORT = 0; Dirección IP y puerto del servidor Kafka.

PRODUCER = 0; Productor de Kafka que permite enviar mensajes.

ID_TAXI = 0; Identificador único del taxi.

posicion_actual = {‘x’: 1, ‘y’: 1}; Indica la posición inicial del taxi.

TAM_MAPA = 20; Tamaño del mapa en el que se mueve el taxi (20 x 20).

MOVIMIENTOS = [(0,1), (0,-1), (1,0), (-1,0), (1,1), (1,-1), (-1,1), (-1,-1)]; Movimientos del taxi.

beforeTaxiState = [“OK”, False]; Ok o KO dependiendo del estado del taxi, y True si lo cambia el sensor, ya que tiene más prioridad, y False si lo cambia la central.

sesnsorActivity = False; Indica si hay actividad en el sensor.

customeError = False; Indica si ha habido un error relacionado con algún cliente.

disconnect = False; Variable que indica una desconexión entre taxi y central, o entre taxi y sensor.

Sensors -> EC_S

Aplicación encargada de simular la funcionalidad de los sensores que tiene un taxi, dentro de un sistema distribuido de taxis con conducción autónoma.

Esta aplicación se encarga de simular incidencias en los taxis mientras sigue un trayecto, como son un semáforo en rojo, un peatón cruzando, un vehículo en medio, una valla o muro enfrente, una obra, un stop e incluso un accidente en la carretera, impidiendo el paso.

Tiene la capacidad de conectarse con Digital Engine, enviando cada segundo el estado actual, “OK” si está todo correcto, o “KO” si se ha producido una incidencia, capturando estas incidencias mediante la pulsación de teclas específicas para cada uno, y solucionando estas con la tecla “espacio”.

La comunicación que se establece con Digital Engine es a través de sockets, conectándose primero mediante un socket a este, y una vez conectado, enviando los mensajes de estado a través de otro socket.

Esta aplicación incluye los siguientes módulos:

conectar_a_DE(ip_DE, puerto_DE)

```
# Función encargada de conectar el sensor con el digital engine
def conectar_a_DE(ip_DE, puerto_DE):
    intentos = 0
    max_intentos = 5

    while intentos < max_intentos:
        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect((ip_DE, int(puerto_DE)))
            print(f"[CONECTADO] Sensor conectado al Digital Engine en {ip_DE}:{puerto_DE}")
            return s
        except ConnectionError:
            intentos += 1
            print(f"[ERROR] No se pudo conectar al Digital Engine en {ip_DE}:{puerto_DE}. Intento {intentos}/{max_intentos}. Reintentando...")
            time.sleep(3)

    print(f"[FALLO] No se pudo establecer conexión después de {max_intentos} intentos. Cerrando sensor.")
    return None
```

Recibe por parámetros la ip y el puerto de digital engine, para establecer una conexión TCP con este, verificando de esta manera que se ha conectado con el, intentándolo hasta 5 veces en el caso en que no se logre conectar en el primer intento.

capturar_tecla()

```
# Función encargada de capturar una tecla, sin tener que pulsar Enter en Windows
def capturar_tecla():
    global TAXI_CAIDO

    if TAXI_CAIDO == True:
        return None

    try:
        if msvcrt.kbhit():
            return msvcrt.getch().decode(FORMAT).lower()

    except Exception:
        print("Tecla no asignada a una incidencia")

    return None
```

No recibe ningún parámetro, y detecta una tecla presionada sin necesidad de presionar "Enter", permitiendo capturar incidencias y liberarlas en tiempo real.

enviar_estado()

```
# Función encargada de enviar el estado a Digital Engine
def enviar_estado(ip_DE, port_DE):
    global DESCONECTADO
    global INCIDENCIA_DETECTADA
    global TIPO_DE_INCIDENCIA
    global TAXI_CAIDO

    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as socket_creado:
            socket_creado.connect((ip_DE, int(port_DE)))
            print(f"[CONECTADO] Conectado a {ip_DE}:{port_DE}")

            while not DESCONECTADO:
                try:
                    estado = KO if INCIDENCIA_DETECTADA else OK

                    if INCIDENCIA_DETECTADA:
                        estado = estado + f", INCIDENCIA: {TIPO_DE_INCIDENCIA}"

                    socket_creado.send(estado.encode(FORMAT))

                    print(f"[ENVIÓ] Estado enviado: {estado}")
                    time.sleep(1)

                except Exception as e:
                    print("Taxi caido")
                    print("Sensor sin taxi, sensor roto, cae sensor")
                    TAXI_CAIDO = True
                    sys.exit(1)

    except ConnectionRefusedError as e:
        print("Taxi sin autenticar, sensor inhabilitado")
        TAXI_CAIDO = True
        sys.exit(1)
```

Recibe por parámetros la ip y el puerto del digital engine para establecer otra conexión TCP con este, pero en este caso, enviándole el estado cada segundo, junto a una incidencia específica en el caso que se detecte una.

detectar_incidencia()

```

# Función encargada de detectar incidencias cuando se presiona una tecla
def detectar_incidencia():
    global DESCONECTADO
    global INCIDENCIA_DETECTADA
    global TIPO_DE_INCIDENCIA
    global TAXI_CAIDO

    while not DESCONECTADO and not TAXI_CAIDO:

        if TAXI_CAIDO == True:
            break

        tecla_capturada = capturar_tecla()

        if tecla_capturada is not None:
            # SEMÁFORO
            if tecla_capturada == 's':
                TIPO_DE_INCIDENCIA = "SEMAFORO EN ROJO"
                INCIDENCIA_DETECTADA = True
            # PERSONA
            elif tecla_capturada == 'p':
                TIPO_DE_INCIDENCIA = "PEATÓN CRUZANDO"
                INCIDENCIA_DETECTADA = True
            # COCHE
            elif tecla_capturada == 'c':
                TIPO_DE_INCIDENCIA = "VEHÍCULO EN MEDIO"
                INCIDENCIA_DETECTADA = True
            # VALLA
            elif tecla_capturada == 'v':
                TIPO_DE_INCIDENCIA = "VALLA ENFRENTE"
                INCIDENCIA_DETECTADA = True
            # MURO
            elif tecla_capturada == 'm':
                TIPO_DE_INCIDENCIA = "MURO ENFRENTE"
                INCIDENCIA_DETECTADA = True
            # OBRA
            elif tecla_capturada == 'o':
                TIPO_DE_INCIDENCIA = "OBRA, CALLE CORTADA"
                INCIDENCIA_DETECTADA = True
            # STOP
            elif tecla_capturada == 't':
                TIPO_DE_INCIDENCIA = "STOP, ANIMALES CRUZANDO"
                INCIDENCIA_DETECTADA = True
            # ACCIDENTE
            elif tecla_capturada == 'a':
                TIPO_DE_INCIDENCIA = "ACCIDENTE DELANTE"
                INCIDENCIA_DETECTADA = True
            # SOLUCIÓN DE INCIDENCIA
            elif tecla_capturada == ' ':
                print("INCIDENCIA SOLUCIONADA")
                INCIDENCIA_DETECTADA = False
            else:
                print('Tecla no asignada a una incidencia')

        if TAXI_CAIDO == True:
            sys.exit(1)

```

main()

```

# Main
def main(ip_DE, port_DE):
    global TAXI_CAIDO

    conexion_digital_engine = conectar_a_DE(ip_DE, port_DE)

    if conexion_digital_engine:

        threadEnviaEstado = threading.Thread(target=enviar_estado, args=(ip_DE, port_DE))
        threadDetectaIncidenca = threading.Thread(target=detectar_incidencia)

        threadEnviaEstado.start()
        threadDetectaIncidenca.start()

        threadEnviaEstado.join()
        threadDetectaIncidenca.join()

        if TAXI_CAIDO == True:
            sys.exit(1)

    else:
        sys.exit(1)

```

Recibe por parámetros la ip y el puerto del digital engine, para inicializar las conexiones y lanzar los hilos necesarios para el envío del estado y para detectar las incidencias.

bloque común en python, encargado de ejecutar el script principal y manejar los parámetros desde consola, es decir, desde la línea de comandos

```

if __name__ == "__main__":
    if len(sys.argv) == 3:
        ip_DE = sys.argv[1]
        port_DE = int(sys.argv[2])

        main(ip_DE, port_DE)

    else:
        print("ERROR!! Falta por poner <IP EC_D> <PUERTO EC_D>")

```

No recibe ningún parámetro, y detecta las incidencias específicas según la tecla pulsada, permite la solución de la incidencia seleccionada mediante la tecla espacio.

Incidencias:

Tecla s: semáforo en rojo

Tecla p: peatón cruzando

Tecla c: vehículo en medio

Tecla v: valla enfrente

Tecla m: muro enfrente

Tecla o: obra, calle cortada

Tecla t: stop, animales cruzando

Esta aplicación utiliza las siguientes librerías y las siguientes variables globales:

```
# Sensors
# Aplicación que simula la funcionalidad de los sensores embarcados en el vehículo

import socket
import time
import threading
import sys
import msvcr

HEADER = 64
FORMAT = 'utf-8'
OK = "OK"
KO = "KO"
DESCONECTADO = False
INCIDENCIA_DETECTADA = False
TIPO_DE_INCIDENCIA = ""
TAXI_CAIDO = False
```

HEADER = 64; Define el tamaño del encabezado en bytes para los mensajes que se envían a través del socket.

FORMAT = 'utf-8'; Define el formato de codificación de caracteres para enviar y recibir mensajes de texto en el socket.

OK = "OK" y **KO = "KO"**; Indican el estado del sensor.

DESCONECTADO = False; Indica si el sensor está desconectado del digital engine, siendo similar en los bucles a while True.

INCIDENCIA_DETECTADA = False; Marca si se ha detectado una incidencia.

TIPO_DE_INCIDENCIA = ""; Almacena la incidencia específica capturada.

TAXI_CAIDO = False; Indica si el taxi está fuera de servicio.

Customers -> EC_Customers

Aplicación encargada de representar a un cliente en el sistema distribuido de taxis.

Su funcionalidad principal es solicitar servicios de taxi, enviar actualizaciones a la central y recibir respuestas sobre el estado de sus solicitudes.

Esta aplicación usa kafka como medio de comunicación entre la central y el cliente para intercambiar mensajes en tiempo real.

Esta aplicación incluye los siguientes módulos:

<sendMessageKafka(topic, msg)>

```
# Envía un mensaje a través de KAFKA
def sendMessageKafka(topic, msg):
    time.sleep(0.5)
    PRODUCER.send(topic, msg.encode(FORMAT))
    PRODUCER.flush()
```

Recibe por parámetros el topic y el mensaje que se va a enviar por kafka a través del topic especificado.

<obtainServices()>

```

# Obtener los servicios del cliente
def obtainServices():
    try:
        ubicacion = 0
        services = []
        with open(f'Requests/EC_Requests_{CUSTOMER_ID}.json', "r", encoding="utf-8") as requests:
            destinos = json.load(requests)

            for destino in destinos["Ubication"]:
                ubicacion = destino["Id"]

            for destino in destinos["Requests"]:
                services.append(destino["Id"])

        return ubicacion, services

    except Exception:
        print(F"NO EXISTE EL ARCHIVO DE SERVICIOS: Requests/EC_Requests_{CUSTOMER_ID}.json")

    return -1, -1

```

No recibe parámetros, y lee los servicios desde un archivo .json específico del cliente. En el caso en el que el archivo no exista, se envía un error.

<executeServices()>

```

# Ejecuta todos los servicios del cliente
def executeServices():
    global disconnect
    ubicacion, services = obtainServices()

    if ubicacion != -1 and services != -1:
        for service in services:
            if disconnect == True:
                print("ERROR!! NO HAY CONEXIÓN CON LA CENTRAL.")
                break

            print(F"\nENVIANDO SIGUIENTE SERVICIO ({service})...")
            completedService = requestService(ubicacion, service)
            if completedService:
                ubicacion = service

            time.sleep(4)

    disconnect = True

```

No recibe parámetros, ejecuta cada servicio solicitado por el cliente, y envía una solicitud a la central para verificar si fue completado o rechazado.

<requestService(ubicacion, destino)>

```

# Solicitud de servicio y mantenimiento de la conexión con central (CUSTOMER STATUS)
def requestService(ubicacion, destino):
    global disconnect
    sendMessageKafka("Customer2Central", f"SOLICITUD DE SERVICIO: {CUSTOMER_ID} {ubicacion} {destino}") # Un taxi primero tiene que ir a la ubicación y luego llevarlo al destino

    consumer = Kafka.KafkaConsumer("Central2Customer", group_id=str(uuid.uuid4()), auto_offset_reset="latest", bootstrap_servers=f"{KAFKA_IP}:{KAFKA_PORT}")

    completedService = False
    declinedService = False
    while not disconnect and not declinedService and not completedService:
        messages = consumer.poll(1000)
        for _, messageValues in messages.items():
            for msg in messageValues:
                msg = msg.value.decode(FORMAT)

                if msg.startswith("CLIENTE"):
                    id2Verify = msg.split(" ")[1]
                    state = msg.split(" ")[3]

                    if id2Verify == CUSTOMER_ID:
                        if state == "ACEPTADO.":
                            print("SERVICIO ACEPTADO Y EN CAMINO...")

                        elif state == "COMPLETADO.":
                            print("SERVICIO COMPLETADO.")
                            completedService = True
                            break

                        else:
                            print("SERVICIO RECHAZADO.")
                            declinedService = True
                            break

                elif msg.startswith("FIN"):
                    disconnect = True
                    break

    consumer.close()
    return completedService

```

Recibe por parámetro la ubicación del cliente y el destino al que quiere ir, enviando a la central la solicitud de servicio esperando una respuesta sobre si es aceptado, completado o rechazado. Además, se desconecta cuando recibe un mensaje FIN.

<showMap()>

```

# Mostrar mapa
def showMap():
    global disconnect

    consumer = kafka.KafkaConsumer("Mapa", group_id=str(uuid.uuid4()), bootstrap_servers=f"{KAFKA_IP}:{KAFKA_PORT}")

    while not disconnect:
        messages = consumer.poll(1000)
        for _, messagesValues in messages.items():
            for msg in messagesValues:
                msg = msg.value.decode(FORMAT)
                print(msg, end="")

```

No recibe ningún parámetro, y muestra actualizaciones en el mapa a medida que las recibe desde Kafka en el topic "Mapa".

<sendCentralActive()>

```

# CUSTOMER AND CENTRAL STATUS
# Envía cada 2 segundos un mensaje a la central para comprobar su actividad
def sendCentralActive():
    while not disconnect:
        sendMessageKafka("Customer2Central", f"ACTIVE?")
        sendMessageKafka("5Status", f"CUSTOMER {CUSTOMER_ID} ACTIVO.")
        time.sleep(2)

```

No recibe parámetros, y envía a la central un mensaje para verificar si esta, está activa.

bloque común en python, encargado de ejecutar el script principal y manejar los parámetros desde consola, es decir, desde la línea de comandos

```

if __name__ == "__main__":
    if (len(sys.argv) == 4):
        KAFKA_IP = sys.argv[1]
        KAFKA_PORT = int(sys.argv[2])
        PRODUCER = kafka.KafkaProducer(bootstrap_servers=f"{KAFKA_IP}:{KAFKA_PORT}")
        CUSTOMER_ID = sys.argv[3]

        main()

    else:
        print("ERROR! Se necesitan estos argumentos: <IP DEL BOOTSTRAP-SERVER> <PUERTO DEL BOOTSTRAP-SERVER> <ID DEL CLIENTE>")

```

<statusControl()>

```

# Recibe el STATUS de la CENTRAL
def statusControl():
    global disconnect
    sendMessageKafka("Status", f"CUSTOMER {CUSTOMER_ID} ACTIVO.")
    consumer = kafka.KafkaConsumer("Central2Customer", group_id=str(uuid.uuid4()), bootstrap_servers=f"{KAFKA_IP}:{KAFKA_PORT}")

    startTime = time.time()
    while not disconnect:
        endTime = time.time()
        if endTime - startTime > 10:
            disconnect = True
            break

        messages = consumer.poll(1000)
        for _, messagesValues in messages.items():
            for msg in messagesValues:
                msg = msg.value.decode(FORMAT)

                if msg == "CENTRAL ACTIVE":
                    startTime = time.time()

    consumer.close()

```

No recibe parámetros, y desconecta al cliente si no recibe una respuesta de actividad de la central en 10 segundos.

<main()>

```

# MAIN
def main():
    threadExecuteServices = threading.Thread(target=executeServices)
    threadStatusControl = threading.Thread(target=statusControl)
    threadCentralActive = threading.Thread(target=sendCentralActive)
    threadMap = threading.Thread(target=showMap)

    print(f"INICIANDO CLIENTE {CUSTOMER_ID}...")
    threadExecuteServices.start()
    threadStatusControl.start()
    threadCentralActive.start()
    threadMap.start()

    return 0

```

No recibe parámetros, es la función principal, e inicia todos los hilos necesarios para ejecutar los servicios y controlar el estado de ejecución.

Esta aplicación utiliza las siguientes librerías y las siguientes variables globales:

```
import sys
import kafka
import threading
import json
import time
import uuid

FORMAT = 'utf-8'
KAFKA_IP = 0
KAFKA_PORT = 0
PRODUCER = 0
CUSTOMER_ID = 0
disconnect = False
```

FORMAT = 'utf-8';

Define el formato de codificación de caracteres para enviar y recibir mensajes de texto en el socket.

KAFKA_IP = 0 y KAFKA_PORT = 0;

Dirección IP y puerto del servidor Kafka.

PRODUCER = 0;

Productor de Kafka que permite enviar mensajes.

CUSTOMER_ID = 0;

Identificador único del cliente.

disconnect = False;

Variable que indica una desconexión entre cliente y central.

Customers -> Requests -> EC_Requests{id_cliente}.json

```
Requests > () EC_Requests_a.json > ...
1  {
2    "Ubication": [
3      {
4        "Id": "B"
5      }
6    ],
7  ],
8
9  "Requests": [
10   {
11     "Id": "A"
12   },
13   {
14     "Id": "C"
15   },
16   {
17     "Id": "F"
18   },
19   {
20     "Id": "D"
21   },
22   {
23     "Id": "E"
24   }
25 ]
26 }
```

```
Requests > () EC_Requests_c.json > ...
1  {
2    "Ubication": [
3      {
4        "Id": "A"
5      }
6    ],
7  ],
8
9  "Requests": [
10   {
11     "Id": "C"
12   },
13   {
14     "Id": "F"
15   },
16   {
17     "Id": "D"
18   },
19   {
20     "Id": "E"
21   },
22   {
23     "Id": "A"
24   }
25 ]
26 }
```

```
Requests > () EC_Requests_d.json > ...
1  {
2    "Ubication": [
3      {
4        "Id": "C"
5      }
6    ],
7  ],
8
9  "Requests": [
10   {
11     "Id": "D"
12   },
13   {
14     "Id": "E"
15   },
16   {
17     "Id": "A"
18   },
19   {
20     "Id": "C"
21   },
22   {
23     "Id": "F"
24   }
25 ]
26 }
```

```

Requests > EC_Requests_e.json > ...
1 { "Ubication": [ { "Id": "D" } ], "Requests": [ { "Id": "E" }, { "Id": "A" }, { "Id": "C" }, { "Id": "F" }, { "Id": "D" } ] }
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Requests > EC_Requests_f.json > ...
1 { "Ubication": [ { "Id": "E" } ], "Requests": [ { "Id": "F" }, { "Id": "D" }, { "Id": "E" }, { "Id": "A" }, { "Id": "C" } ] }
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Requests > EC_Requests.json > ...
1 { "Requests": [ { "Id": "A" }, { "Id": "C" }, { "Id": "F" }, { "Id": "D" }, { "Id": "E" } ] }
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

```

Podemos observar 5 clientes, con la ubicación en la que se encuentra, siendo esta una localización de las existentes, ya que la lógica de nuestra aplicación es que el cliente tiene que estar en la ubicación de la localización que actúa como una parada de taxis.

Y, además de la ubicación, se encuentran los servicios con las localizaciones a las que quiere ir, en orden.

La última foto que aparece, es el ejemplo puesto como guía sobre cómo tiene que ser este fichero .json.

Locations -> EC_locations.json

```
{  
    "locations": [  
        {  
            "Id": "A",  
            "POS": "5,3"  
        },  
        {  
            "Id": "B",  
            "POS": "2,11"  
        },  
        {  
            "Id": "C",  
            "POS": "8,16"  
        },  
        {  
            "Id": "D",  
            "POS": "17,16"  
        },  
        {  
            "Id": "E",  
            "POS": "19,10"  
        },  
        {  
            "Id": "F",  
            "POS": "11,5"  
        }  
    ]  
}
```

Fichero .json, que incluye las localizaciones a las que puede acceder el taxi, pudiendo añadir o eliminar una de estas desde este fichero, o incluso editarlo cambiando el id o la coordenada en la que se encuentra.

Como se puede observar, cada localización tiene un id, representado por una letra en mayúscula junto a una posición, que indica la coordenada en la que se encuentra en el mapa.

Base de datos -> database.txt

Siguiendo el siguiente formato:

<id_taxi>,<destino_taxi>,<activo/inactivo_taxi>.<servicio_taxi>

```
1,-,NO.Parado  
2,-,NO.Parado  
3,-,NO.Parado  
4,-,NO.Parado  
5,-,NO.Parado  
6,-,NO.Parado  
7,-,NO.Parado  
8,-,NO.Parado  
9,-,NO.Parado  
10,-,NO.Parado
```

<id_taxi> -> entero que representa el id del taxi
<destino_taxi>:
 -> localización
 -> “-”: ningún destino
 -> “base”: volver a la base (1,1)
<activo/inactivo_taxi>:
 -> NO: sin autenticar
 -> OK: autenticado con estado OK
 -> KO: autenticado con estado KO (incidencia)
<servicio_taxi>:
 -> Parado: taxi parado
 -> Servicio Central: servicio solicitado por la central
 -> Servicio {id_cliente}: servicio solicitado por el cliente

Guía de despliegue de la aplicación.

Para el despliegue de la aplicación, se han creado tres carpetas, las cuales se llaman PC1, PC2 y PC3, donde se encuentra lo necesario para desplegar la aplicación en cada PC, tal y como se indica en el nombre de la carpeta.

Se va a desplegar tal y como pone en el enunciado:



Estas carpetas, se encuentran en un pendrive para poder desplegar más rápido la aplicación, y por si surge algún error con respecto al pendrive, también se encuentran en un disco duro externo, y en el caso que este falle, además, se encuentran en un repositorio de GitHub.

Antes de realizar nada, en los tres pc se va a desinstalar python 12, y a instalar python 11, ya que python 12 da un error con la librería de kafka-python, que no se ha solucionado actualmente.

Una vez realizado esto en los tres, se procede a desplegar la aplicación.

Además de las tres carpetas para los respectivos PCs, hay un fichero pipInstall.bat, encargado de instalar las librerías necesarias para el despliegue de nuestra aplicación. De manera que este fichero se va a ejecutar en los tres PCs.

En el PC1, se va a abrir la carpeta PC1, la cual contendrá una carpeta llamada "Control de kafka", donde se encuentra la carpeta de "kafka", la cual hay que copiarla y pegarla en el disco C: del pc.

Una vez hecho esto, dentro de "Control de kafka", hay tres .bat, siendo uno "startKafka.bat", encargado de iniciar zookeeper y kafka, otro "createTopics.bat", encargado de crear los topics necesarios para el despliegue de la aplicación, y un último "stopKafka.bat", siendo el encargado de parar zookeeper y kafka una vez finalizado el despliegue de la práctica.

En esta carpeta, también se encuentra el .exe de customer, donde se podrán ejecutar tantos clientes como se quiera, una vez ejecutada la central y los taxis, para que a los clientes se les asigne un servicio. Además, en esta carpeta, se encuentra un carpeta Requests, con los requests.json de los clientes, llamados "EC_Requests_{id_cliente}.json".

En el PC2, se va a abrir la carpeta PC2, la cuál contendrá el .exe de la central, siendo el primero en ejecutarse, junto con un fichero llamado "database.txt", actuando como base de datos, y junto con un fichero llamado "EC_locations.json", donde se encuentran las localizaciones.

En el PC3, se va a abrir la carpeta PC3, la cuál contendrá el .exe tanto del digital engine como del sensor, ejecutándose estos una vez ya se ha ejecutado la central.

Capturas de pantalla mostrando el funcionamiento de la aplicación.

```
Windows PowerShell
```

```
PS C:\Users\Javier UA\easyCab_sd> .\EC_Central.exe
ERROR! Se necesitan estos argumentos: <IP DE ESCUCHA> <PUERTO DE ESCUCHA> <IP DEL BOOTSTRAP-SERVER> <PUERTO DEL BOOTSTRAP-SERVER>
PS C:\Users\Javier UA\easyCab_sd> .\EC_Central.exe 192.168.1.106 5050 192.168.56.1 9092
INICIANDO CENTRAL...
CENTRAL A LA ESCUCHA EN <socket.socket fd=688, family=2, type=1, proto=0, laddr=('192.168.1.106', 5050)>

-----
*** EASY CAB Release 1 ***
-----
Taxis | Clientes
-----|-----
Id. Destino Estado | Id. Destino Estado
-----|-----
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
-----
1 . . . . . . . . . . . . . .
2 . . . . . . . . . . . . . .
3 . . . . . . . . . . . . . .
4 . . . . . . . . . . . . . .
5 . . A . . . . . . . . . . .
6 . . . . . . . . . . . . . .
7 . . . . . . . . . . . . . .
8 . . . . . . . . . . . . . .
9 . . . . . . . . . . . . . .
10 . . . . . . . . . . . . . .
11 . . F . . . . . . . . . . .
12 . . . . . . . . . . . . . .
13 . . . . . . . . . . . . . .
14 . . . . . . . . . . . . . .
15 . . . . . . . . . . . . . .
16 . . . . . . . . . . . . . .
17 . . . . . . . . . . . . . .
18 . . . . . . . . . . . . . .
19 . . . . . . . . . . . . . .
20 . . . . . . . . . . . . . .

-----
Estado general del sistema: OK
```



```

Windows PowerShell
18 . . . . . E . . . . . 5
19 . . . . . . . . . .
20 . . . . . . . . . .
Estado general del sistema: OK

*** EASY CAB Release 1 ***
Taxis | Clientes
-----|-----
Id. Destino Estado | Id. Destino Estado
4 B OK. Servicio a | a A OK. Taxi 4
5 D OK. Servicio e | e E OK. Taxi 5
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
1 . . . . . 4 . . . . . a . . . . .
2 . . . . . . . . . .
3 . . . . . . . . . .
4 . . . . . . . . . .
5 . . . . . . . . . .
6 . . . . . . . . . .
7 . . . . . . . . . .
8 . . . . . . . . . .
9 . . . . . . . . . .
10 . . . . . . . . . .
11 . . . . . . . . . .
12 . . . . . . . . . .
13 . . . . . . . . . .
14 . . . . . . . . . .
15 . . . . . . . . . .
16 . . . . . . . . . .
17 . . . . . . . . . .
18 . . . . . . . . . .
19 . . . . . . . . . .
20 . . . . . . . . . .
Estado general del sistema: OK
|
```



```

Windows PowerShell
16 . . . . . . . . . .
17 . . . . . . . . . .
18 . . . . . . . . . .
19 . . . . . . . . . .
20 . . . . . . . . . .
Estado general del sistema: OK

*** EASY CAB Release 1 ***
Taxis | Clientes
-----|-----
Id. Destino Estado | Id. Destino Estado
4 B OK. Servicio a | a A OK. Taxi 4
5 D OK. Servicio e | e E OK. Taxi 5
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
1 . . . . . . . . . .
2 . . . . . . . . . .
3 . . . . . . . . . .
4 . . . . . . . . . .
5 . . . . . . . . . .
6 . . . . . . . . . .
7 . . . . . . . . . .
8 . . . . . . . . . .
9 . . . . . . . . . .
10 . . . . . . . . . .
11 . . . . . . . . . .
12 . . . . . . . . . .
13 . . . . . . . . . .
14 . . . . . . . . . .
15 . . . . . . . . . .
16 . . . . . . . . . .
17 . . . . . . . . . .
18 . . . . . . . . . .
19 . . . . . . . . . .
20 . . . . . . . . . .
Estado general del sistema: OK
|
```



```

Windows PowerShell
18 . . . . . E . . . . . 5e . . . . .
19 . . . . . . . . . .
20 . . . . . . . . . .
Estado general del sistema: OK

*** EASY CAB Release 1 ***
Taxis | Clientes
-----|-----
Id. Destino Estado | Id. Destino Estado
4 A OK. Servicio a | a A OK
5 E OK. Servicio e | e E OK
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
1 . . . . . . . . . .
2 . . . . . . . . . .
3 . . . . . . . . . .
4 . . . . . . . . . .
5 . . . . . . . . . .
6 . . . . . . . . . .
7 . . . . . . . . . .
8 . . . . . . . . . .
9 . . . . . . . . . .
10 . . . . . . . . . .
11 . . . . . . . . . .
12 . . . . . . . . . .
13 . . . . . . . . . .
14 . . . . . . . . . .
15 . . . . . . . . . .
16 . . . . . . . . . .
17 . . . . . . . . . .
18 . . . . . . . . . .
19 . . . . . . . . . .
20 . . . . . . . . . .
Estado general del sistema: OK
|
```



```

Windows PowerShell
16 . . . . . . . . . .
17 . . . . . . . . . .
18 . . . . . . . . . .
19 . . . . . . . . . .
20 . . . . . . . . . .
Estado general del sistema: OK

*** EASY CAB Release 1 ***
Taxis | Clientes
-----|-----
Id. Destino Estado | Id. Destino Estado
4 A OK. Servicio a | a A OK
5 E OK. Servicio e | e E OK
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
1 . . . . . . . . . .
2 . . . . . . . . . .
3 . . . . . . . . . .
4 . . . . . . . . . .
5 . . . . . . . . . .
6 . . . . . . . . . .
7 . . . . . . . . . .
8 . . . . . . . . . .
9 . . . . . . . . . .
10 . . . . . . . . . .
11 . . . . . . . . . .
12 . . . . . . . . . .
13 . . . . . . . . . .
14 . . . . . . . . . .
15 . . . . . . . . . .
16 . . . . . . . . . .
17 . . . . . . . . . .
18 . . . . . . . . . .
19 . . . . . . . . . .
20 . . . . . . . . . .
Estado general del sistema: OK
|
```

```
Windows PowerShell x + - x Windows PowerShell x Windows PowerShell x + - x
17 . . . . . D . . .
18 . . . . . 5 . . .
19 . . . . . . . . .
20 . . . . . . . . .

Estado general del sistema: OK
-----
```



```
Windows PowerShell x + - x Windows PowerShell x Windows PowerShell x + - x
16 . . . . . 0 . . .
17 . . . . . e . . .
18 . . . . . . . . .
19 . . . . . . . . .
20 . . . . . . . . .

Estado general del sistema: OK
-----
```



```
*** EASY CAB Release 1 ***
-----
```

Taxis			Clientes		
Id.	Destino	Estado	Id.	Destino	Estado
4	A NO.	Servicio a	a	A	OK
5	A OK.	Servicio e	e	A	OK


```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```



```
1 . . . . . B . . .
2 . . . . . . . . .
3 . . . . . . . . .
4 . . . . . . . . .
5 . . A . . . 4! . .
6 . . . . . . . . .
7 . . . . . . . . .
8 . . . . . . . . .
9 . . . . . . . . .
10 . . . . . . . . .
11 . . . . . F . . .
12 . . . . . . . . .
13 . . . . . . . . .
14 . . . . . . . . .
15 . . . . . . . . .
16 . . . . . . . . .
17 . . . . . . . . .
18 . . . . . . . . .
19 . . . . . . . . .
20 . . . . . . . . .

Estado general del sistema: OK
-----
```



```
*** EASY CAB Release 1 ***
-----
```

Taxis			Clientes		
Id.	Destino	Estado	Id.	Destino	Estado
4	A NO.	Servicio a	a	A	OK
5	E OK.	Servicio e	e	A	OK.
		Taxi 5			


```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```



```
1 . . . . . . . . .
2 . . . . . . . . .
3 . . . . . . . . .
4 . . . . . . . . .
5 . . A . . . 4! . .
6 . . . . . . . . .
7 . . . . . . . . .
8 . . . . . . . . .
9 . . . . . . . . .
10 . . . . . . . . .
11 . . . . . F . . .
12 . . . . . . . . .
13 . . . . . . . . .
14 . . . . . . . . .
15 . . . . . . . . .
16 . . . . . . . . .
17 . . . . . . . . .
18 . . . . . . . . .
19 . . . . . . . . .
20 . . . . . . . . .

Estado general del sistema: OK
-----
```