UA Universitat d'Alacant
Universidad de Alicante

# Sistemas distribuidos

## Practica 1

Carlos Francisco Campos Lezameta 50383440P
Jorge Manuel Aquino Sanchez 55336161R
Año 2025/2026

# 1. Introducción

El proyecto consiste en la simulación de una red de puntos de recarga de vehículos eléctricos, gestionada por una central de control que supervisa, autoriza y monitoriza el funcionamiento de cada estación, así como las solicitudes de recarga enviadas por los conductores a través de sus aplicaciones. La comunicación entre los distintos módulos (Central, Charging Points y Drivers) se realiza mediante sockets TCP y un sistema de mensajería basado en Kafka, garantizando un intercambio de datos en tiempo real y de manera asíncrona.

Esta práctica tiene como objetivo aplicar los conceptos sobre comunicación mediante sockets, streaming de eventos, colas de mensajería y modularidad en el desarrollo de un sistema distribuido realista.

# 2. Componentes de la práctica

## a. Central

```python
def main():

    print("****** EV_Central ******")

    # Pedir la IP al usuario
    SERVER = input("Introduce la IP del servidor: ").strip()
    print(f"[INFO] Servidor configurado en {SERVER}:{PORT}\n")
    ADDR = (SERVER, PORT)

    clientes = cargarClientes(CLIENTES_FILE)
    i = 0
    for c in clientes:
        if i%2 == 0:
            CUSTOMER_IDX.append(c)
        i+=1

    bootstrap = SERVER
    t_kafka = threading.Thread(target=run_kafka_loop, args=(bootstrap,), daemon=True)
    t_kafka.start()

    CPS, CPS_IDX = cargarCPs(CPS_FILE)

    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)  # opcional pero útil
    server.bind(ADDR)

    print("[STARTING] Servidor inicializándose...")
    start(server)

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

En la función principal de la Central se pide introducir la **ID** que se usará para crear el **SOCKET** que mantendrá la comunicación con los **monitores**, así como los **TOPICs** que se utilizarán para la comunicación entre los distintos **DRIVERS** y **ENGINES**.

También se cargan los **ID de los clientes** presentes en la base de datos, entre otras cosas, para que, si un cliente desea autenticarse, se pueda comprobar si el identificador ya está siendo utilizado por otro driver o no. Además, se cargan desde la base de datos los **identificadores de los puntos de recarga (CPs)**.

Como se puede observar en la imagen, se crea un **hilo** por cada **DRIVER** o **ENGINE** que intente conectarse mediante **Kafka** a la central.
En el caso de los **monitores**, la central queda a la espera de una conexión; una vez esta ocurre, se atiende a cada uno de ellos en un hilo independiente, como se mostrará más adelante en los métodos auxiliares.

La función **start**, presente en la función principal del programa, cuenta con la siguiente definición:

```
def start(server):
    global CONEX_ACTIVAS
    server.listen()
    print(f"[LISTENING] Servidor a la escucha en {SERVER}")
    print(f"[CONEXIONES ACTIVAS] {CONEX_ACTIVAS}")
    while True:
        conn, addr = server.accept()
        CONEX_ACTIVAS += 1
        if (CONEX_ACTIVAS <= MAX_CONEXIONES):
            msg = "1"
            conn.send(msg.encode(FORMAT)) # Enviamos un 1 para informar que nos conectamos correctamente
            thread = threading.Thread(target=handle_client, args=(conn, addr))
            thread.start()
            print(f"[CONEXIONES ACTIVAS] {CONEX_ACTIVAS}")
            print("CONEXIONES RESTANTES PARA CERRAR EL SERVICIO", MAX_CONEXIONES-CONEX_ACTIVAS)
        else:
            print("OOppsss... DEMASIADAS CONEXIONES. ESPERANDO A QUE ALGUIEN SE VAYA")
            conn.send("OOppsss... DEMASIADAS CONEXIONES. Tendrás que esperar a que alguien se vaya".encode(FORMAT))
            conn.close()
```

Como podemos observar, se utiliza una variable global llamada **CONEX_ACTIVAS** para controlar el número de monitores que pueden conectarse a la central. Nunca se pueden conectar más de **MAX_CONEXIONES**, valor que está fijado en **dos** dentro del programa.

Si, al aceptar una nueva conexión, el número de conexiones activas **no supera el máximo permitido**, se envía un mensaje de confirmación al monitor (un **1**), y se crea un **hilo** para dicha conexión, la cual será atendida en la función **handle_client**, de la cual hablaremos a continuación.

```
def handle_client(conn, addr):
    print(f"[NUEVA CONEXION] {addr} connected.")

    connected = True
    msg_aux = ""  # Lo usamos para mostrar el mensaje del cliente una vez
    while connected:
        try:
            msg_length = conn.recv(HEADER).decode(FORMAT)
            if msg_length:
                msg_length = int(msg_length)
                msg = conn.recv(msg_length).decode(FORMAT)
                if msg == FIN:
                    connected = False
                else:
                    parts = msg.split("|")  # split quita los separadores y convierte el mensaje en una lista
                    if parts[0] == "monitor":
                        resp = attendToMonitor(parts[1], parts[2], parts[3])  # LLAMAR A LA FUNCION attendToMonitor
                        conn.send(resp.encode(FORMAT))
                    else:
                        resp = "ERROR: peticion de origen desconocido"
                if msg != msg_aux:
                    print(f" He recibido del cliente [{addr}] el mensaje: {msg}")
                msg_aux = msg

        except ConnectionResetError:
            print(f"[{addr}] Conexión interrumpida por el cliente (WinError 10054).")
            break  # salimos del bucle si el cliente se desconecta abruptamente

    print("ADIOS. TE ESPERO EN OTRA OCASION")
    conn.close()
```

La función **handle_client** es la que atiende cada aspecto de una conexión. Como podemos observar, solo espera que se conecten **monitores** a la central mediante **sockets**, y lo hace dentro de un **bucle** que permanece activo

```
def handle_client(conn, addr):
    print(f"[NUEVA CONEXION] {addr} connected.")

    connected = True
    msg_aux = ""  # Lo usamos para mostrar el mensaje del cliente una vez
    while connected:
        try:
            msg_length = conn.recv(HEADER).decode(FORMAT)
            if msg_length:
                msg_length = int(msg_length)
                msg = conn.recv(msg_length).decode(FORMAT)
                if msg == FIN:
                    connected = False
                else:
                    parts = msg.split("|")  # split quita los separadores y convierte el mensaje en una lista
                    if parts[0] == "monitor":
                        resp = attendToMonitor(parts[1], parts[2], parts[3])  # LLAMAR A LA FUNCION attendToMonitor
                        conn.send(resp.encode(FORMAT))
                    else:
                        resp = "ERROR: peticion de origen desconocido"
                if msg != msg_aux:
                    print(f" He recibido del cliente [{addr}] el mensaje: {msg}")
                msg_aux = msg

        except ConnectionResetError:
            print(f"[{addr}] Conexión interrumpida por el cliente (WinError 10054).")
            break  # salimos del bucle si el cliente se desconecta abruptamente

    print("ADIOS. TE ESPERO EN OTRA OCASION")
    conn.close()
```

recibiendo peticiones hasta que el monitor que estableció la conexión con la central **cierra la conexión**.

El mecanismo de la función es el siguiente: espera un mensaje con el formato **"monitor|PETICION|TERCERA_COMPONENTE"**, siendo el último elemento del mensaje el **estado** o **localización** del **CP**.

Se comprueba si la cabecera del mensaje es "monitor"; si lo es, se llama a la función **attendToMonitor()**. En caso contrario, se trata como una **situación de error**, y no se atiende la petición por ser de **origen desconocido**.

Dentro de la función **attendToMonitor()** se atienden las dos peticiones posibles de un monitor: **"AUTENTIFICACION"** y **"ESTADO"**.
Con la primera, la central intenta insertar un nuevo **CP** en la base de datos si el identificador es distinto de los ya presentes, y con la segunda, actualiza el **estado** de un **CP** ya registrado en la BD.

A continuación, se muestra su implementación:

```python
def attendToMonitor(peticion, id_cp, var):
    """
    peticion: 'AUTENTIFICACION' | 'ESTADO'
    """
    if peticion == "AUTENTIFICACION":
        # ¿ya está en memoria?
        if id_cp in CPS_IDX:
            return "central|ERROR"

        if not insertToCPsBD(id_cp, var, "IDLE"):
            return "central|ERROR"

        CPS_IDX.append(id_cp)
        return "central|OK"

    elif peticion == "ESTADO":
        if id_cp not in CPS_IDX:
            return "central|ERROR"

        if not updateStatusCP(id_cp, var):
            return "central|ERROR"

        return "central|OK"

    else:
        if id_cp in CPS_IDX:
            return "REGISTRADO"
        return "DESCONOCIDO"
```

Por otra parte, la central atiende las peticiones procedentes del Engine o del Driver mediante Kafka.

El núcleo del funcionamiento de este mecanismo de comunicación se encuentra dentro de la función run_kafka_loop(), cuya definición se muestra a continuación.

```python
def run_kafka_loop(bootstrap):
    """Crea consumer/producer y entra al bucle Kafka (hilo dedicado)."""
    consumer = None
    producer = None
    try:
        consumer = create_consumer(bootstrap)
        producer = create_producer(bootstrap)
        receive_messages(consumer, producer)   # bucle bloqueante
    except Exception as e:
        print(f"[CENTRAL] Hilo Kafka terminado con error: {e}")
    finally:
        try:
            if consumer is not None:
                consumer.close()
        except Exception:
            pass
        try:
            if producer is not None:
                producer.flush()
                producer.close()
        except Exception:
            pass
```

En esta función se crean los objetos consumer y producer, que permiten recibir y enviar mensajes a través de los topics de Kafka.

El parámetro bootstrap hace referencia a la dirección del clúster Kafka (por ejemplo, una IP o un hostname con su puerto, como 192.168.1.10:9092).

Dicho valor indica a la central dónde se encuentra el broker principal de Kafka, de manera que pueda establecer la conexión y unirse al sistema de mensajería distribuida.

En caso de que se produzca un error durante la ejecución del hilo Kafka, la función se encarga de cerrar correctamente las conexiones de consumer y producer antes de finalizar.

A continuación se muestra también las definiciones de las funciones create_consumer y create_producer, para un mayor entendimiento de su funcionamiento.

```python
def create_producer(bootstrap):
    """
    Crea un productor Kafka con la configuración básica.
    """
    try:
        producer = KafkaProducer(
            bootstrap_servers=[bootstrap],
            value_serializer=lambda s: s.encode(FORMAT),
            linger_ms=10,
        )
        print(f"[CENTRAL] Productor conectado a {bootstrap}")
        return producer
    except Exception as e:
        print(f"[CENTRAL] No puedo crear el productor en '{bootstrap}': {e}")
        print("--> Verifica que Kafka está arrancado, el puerto es 9092 y advertised.listeners es accesible.")
        raise
```

```python
def create_consumer(bootstrap):
    try:
        consumer = KafkaConsumer(
            TOPIC_REQUESTS, TOPIC_CENTRAL,
            bootstrap_servers=[bootstrap],
            value_deserializer=lambda m: m.decode(FORMAT),
            auto_offset_reset="earliest",
            enable_auto_commit=True,
            group_id="central-group",
            client_id="central-1",
        )
        print(f"[CENTRAL] Conectado a {bootstrap}, escuchando '{TOPIC_REQUESTS}'…")
        return consumer
    except Exception as e:
        print(f"[CENTRAL] No puedo conectar con el broker '{bootstrap}': {e}")
        print("--> Verifica que Kafka está arrancado, el puerto es 9092 y advertised.listeners es accesible.")
        raise
```

Por supuesto, es imperativo explicar el funcionamiento de la función **receive_messages()**, presente también en **run_kafka_loop()**, y cuya definición se muestra a continuación:

```
def receive_messages(consumer, producer):
    for msg in consumer:
        text = msg.value or ""
        parts = text.split("|")

        if len(parts) == 3:
            # driver|PETICION|DRIVER_ID
            remitente = parts[0].lower()
            peticion  = parts[1]
            driver_id = parts[2]
            cp_id     = ""
        elif len(parts) == 4:
            # driver|PETICION|CP_ID|DRIVER_ID   o   engine|RESPUESTA|CP_ID|DRIVER_ID
            remitente = parts[0].lower()
            peticion  = parts[1]
            cp_id     = parts[2]
            driver_id = parts[3]
        else:
            print(f"[CENTRAL] Formato inválido: {text}")
            continue

        print(f"[CENTRAL] Recibido: {text}")
        print(f"  --> remitente: {remitente}, Driver: {driver_id}, CP: {cp_id}")

        if remitente == "driver":
            # tu attendToDriver ahora recibe también el producer
            attendToDriver(peticion, cp_id, driver_id, producer)

        elif remitente == "engine":
            # el engine habla como "engine|...", pero tu attendToEngine
            # quiere "driver|..." -> lo adaptamos aquí sin cambiar tu lógica
            if peticion == FIN:
                print(f"[CENTRAL] Driver {driver_id} cerró sesión.")
            msg_txt = f"driver|{peticion}|{driver_id}"
            attendToEngine(producer, msg_txt)
```

Dentro de esta función se reciben los mensajes que llegan a través del **consumer** de Kafka y se **analiza quién es el remitente** de cada uno.
En función de la cabecera de la petición, se **discrimina el origen del mensaje** (ya sea un **Driver** o un **Engine**) y se **delegan las acciones correspondientes** a las funciones **attendToDriver()** o **attendToEngine()**.

El mensaje se descompone utilizando el carácter `'|'` como separador, y dependiendo del número de componentes se interpreta como una **petición** o una **respuesta**.
En caso de recibir un formato incorrecto o incompleto, la función lo detecta y muestra un mensaje de error informativo sin procesar la petición.

De esta forma, **receive_messages()** actúa como un **intermediario central** dentro del sistema de mensajería Kafka, garantizando que cada mensaje se enrute correctamente al módulo encargado de su tratamiento.

Por último, es importante explicar las funciones **attendToEngine()** y **attendToDriver()**, ya que son las encargadas de **gestionar las peticiones y respuestas** entre los distintos componentes del sistema.

```python
def attendToDriver(peticion, cp_id, driver_id, producer=None):
    """
    - AUTENTIFICACION: lo gestiona Central (Clientes.txt) -> "central|OK"/"central|ERROR".
    - AUTORIZACION y ESTADO: si el CP existe en CPS_IDX, reenvía al Engine; si no, "central|ERROR".
    Devuelve string breve para logs.
    """
    if peticion == "AUTENTIFICACION":
        if addCustomer(driver_id):
            print("AUTENTIFICACION OK")
            return "central|OK"
        else:
            print("AUTENTIFICACION ERROR")
            return "central|ERROR"

    if peticion in ("AUTORIZACION", "ESTADO", "FIN"):
        if cp_id not in CPS_IDX:
            print(f"{peticion}: ERROR (CP '{cp_id}' no encontrado en Central)")
            return "central|ERROR"
        else:
            print(f"{peticion}: reenviado a Engine (cp={cp_id}, driver={driver_id})")
            if producer is not None:
                replyToEngine(producer, peticion, cp_id, driver_id)
            return "central|OK"

    print(f"Petición no soportada en Central: {peticion}")
    return "central|ERROR"
```

En esta función se procesan las **peticiones provenientes del Driver**.
Si la petición corresponde a una **AUTENTIFICACIÓN**, la central la gestiona directamente comprobando si el identificador del driver ya existe en la base de datos. En caso contrario, el nuevo driver se añade y se devuelve una respuesta de confirmación al solicitante.

Por otro lado, si la petición no es de autenticación (por ejemplo, **AUTORIZACIÓN**, **ESTADO** o **FIN**), la central verifica que el **CP** solicitado exista en su lista de puntos de recarga (**CPS_IDX**).
Si existe, la petición se reenvía al **Engine** a través del **producer Kafka**; si no, se devuelve un mensaje de error indicando que el CP no se encuentra registrado.
De este modo, **attendToDriver()** actúa como intermediario entre los drivers y los engines, asegurando que solo se procesen peticiones válidas y que las comunicaciones estén correctamente encaminadas.

```python
def attendToEngine(producer, msg_txt: str):
    """
    Procesa una respuesta que viene del Engine (por Kafka) y la reenvía al driver.
    Formatos esperados:
        - "driver|120|DRIVER_ID"     -> 120 segundos
        - "driver|IDLE|DRIVER_ID"    -> estado del CP

    Siempre reenvía con:
        "central|...|DRIVER_ID"
    usando replyToDriver(producer, texto)
    """

    parts = msg_txt.split("|")

    # Mensaje mínimo: driver|respuesta|DRIVER_ID
    if len(parts) != 3:
        replyToDriver(producer, "central|ERROR")
        return

    origen = parts[0]
    respuesta = parts[1]
    driver_id = parts[2]

    # 1) Caso TIEMPO --> si la respuesta es un número entero
    if respuesta.isdigit():
        segundos = int(respuesta)
        horas = segundos / 3600
        precio = horas * PRICE_PER_KWH
        resp = f"TIEMPO={segundos}s;PRECIO={precio:.2f}€|{driver_id}"
        replyToDriver(producer, resp)
        return

    # 2) Caso ESTADO → si no es número, lo tratamos como texto de estado
    resp = f"central|{respuesta}|{driver_id}"
    replyToDriver(producer, resp)
    return
```

En esta función se procesan las respuestas que provienen del Engine tras haber atendido la petición enviada por un driver.

La central recibe dichas respuestas por Kafka ylas reenvía al driver correspondiente, utilizando la función replyToDriver().

Los mensajes pueden contener tanto información numérica, como el tiempo de carga restante o el coste calculado en función del consumo, como también estados textuales del punto de recarga.

Si el mensaje recibido tiene un formato incorrecto o incompleto, la central devuelve una respuesta de error al driver.

En definitiva, **attendToEngine()** completa el ciclo de comunicación iniciado por el driver, garantizando que cada respuesta del Engine llegue al destino adecuado con el formato correcto.

## b. Monitor

El monitor es uno de los componentes principales de la practica ya que su función principal es supervisar el estado operativo de cada punto de recarga y garantizar la comunicación constante con la central de control y con el módulo Engine del propio punto.

```
#conecta con central
def conectar_central(self):
    print("[DEBUG] Creando socket del Monitor...")
    self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.sock.connect(self.addr)

    msg=f"monitor|PETICION|{self.ID}|IDLE"
    send(msg, self.sock)

    response=self.sock.recv(2048).decode(FORMAT)
    #mensaje de confirmacion de central
    print(f"[DEBUG] Respuesta recibida: '{response}'")

    if "DEMASIADAS CONEXIONES" in response.upper():
        print("El servidor central está lleno. Espera un momento e inténtalo de nuevo.")
        self.sock.close()
        return False
    if int(response) == 1:
        print("conectado con central")

        #respuesta
        msg_length=self.sock.recv(HEADER).decode(FORMAT)
        #El CP se registra a la base de datos o ya esta registrado
        if msg_length == "DESCONOCIDO":
            msg=f"monitor|AUTENTIFICACION|{self.ID}|{self.LOC}"
            send(msg, self.sock)
            status=self.sock.recv(2048).decode(FORMAT)
            print(status)
            return True
        elif msg_length=="REGISTRADO":
            return True
        else:
            return False
```

Cuando el monitor se enciende, lo primero que hace es conectarse con la central se autentifica usando su identificador como punto de carga y verifica que está listo para prestar sus servicios, si el punto de carga no está registrado en la central este lo guarda en la base de datos.

```python
def estado(self):  #cliente
    client_engine= socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_engine.settimeout(8)
    try:
        client_engine.connect(self.addr_engine)
        print ("Conexion establecida con engine")
        while True:

            msg_stat=f"ENGINE|{self.ID}|ok"
            send(msg_stat, client_engine)
            status=client_engine.recv(2048).decode(FORMAT)
            time.sleep(1)

            #el engine activa el boton KO o no funciona
            if(status == "ENGINE|ERROR"):

                msg_stat=f"monitor|ESTADO|{self.ID}|ERROR"
                send(msg_stat, self.sock)

                print("Averia reportada")
                break
                #El engine esta en uso
            elif (status=="ENGINE|CHARGING"):

                msg_stat=f"monitor|ESTADO|{self.ID}|CHARGING"
                send(msg_stat, self.sock)
                #El engine no envia mas mensajes por lo tanto esta cerrado
            elif not status:
                print("Se cerro la conexion")
                msg_stat=f"monitor|ESTADO|{self.ID}|ERROR"
                send(msg_stat, self.sock)
                  #El estado funciona perfectamente y no esta en uso
            else:
                msg_stat=f"monitor|ESTADO|{self.ID}|IDLE"
                send(msg_stat, self.sock)
```

Al acabar este proceso, establece una conexión mediante sockets con el engine del punto de carga, y le envía mensajes periódicamente para comprobar el estado de esta, para después mandarlo a la central para que lo guarde en la base de datos. Esta monitorización continua permite detectar en tiempo real posibles averías, desconexiones o fallos de comunicación. En caso de que haya un error o  no recibe respuestas del engine, este se lo notifica directamente a central, que actualizará su estado a averiado.

### c. Engine

El componente Engine representa el núcleo del punto de carga, siendo responsable directo de ejecutar las operaciones de suministro eléctrico y de comunicarse con la central de control y con su monitor correspondiente. Al iniciar la ejecución del engine, lo primero que se hace es conectar con el monitor para indicar su presencia a central como punto de carga activo.

```python
def servicios(self):
    try:
        for message in self.consumer:
            text = message.value or ""
            parts = text.split("|")
            peticion = parts[1]
            cp_id     = parts[2]
            driver_id = parts[3]
            self.driver=parts[3]
            print("[Mensaje recibido de central]")
            print(f"[Mensaje del driver: {text}]")
            #Me dice de enchufar central
            if peticion == "AUTORIZACION":
                self.enchufar(cp_id, driver_id)
            #Me dice que lo desenchufe del driver
            elif peticion == "FIN":
                self.desenchufar(driver_id, cp_id)

            elif peticion == "ESTADO":
                self.estado_driver(driver_id, cp_id)
            elif peticion == "APAGAR":
                self.boton_ko
            else:
                print("[Peticion no identificada]")
    except AssertionError:
        print("El engine se desactivo")
```

Tras eso se conecta con el gestor de colas de eventos (kafka) y queda a la espera de recibir solicitudes de servicio procedentes de la central. Esta solicitudes pueden ser:

- Autorización: El driver solicita un servicio de carga con el que conectara el coche con con el punto de carga
- Fin: El driver solicita al punto de carga desenchufar su coche, al desenchufar el coche, el engine mandará a central el tiempo que ha estado el coche cargando en el engine y este a su vez se lo mandara al driver.
- Estado: El driver pedirá una consulta del estado en el que está un punto de carga.
- Apagar: Esta opción es la que mandara central al punto de carga para que se desactive.

```python
def boton_ko(self):
    print("Boton KO presionado")
    if self.status == "CHARGING":
        time_end=time.time() - self.start_time
        print(f"Engine: Carga detenida. Tiempo: {time_end:.2f} seg")
        start_time=None
        mensaje=(f"engine|{time_end:.2f}|{self.driver}")
        self.producer.send(TOPIC_CENTRAL, value=mensaje)
        self.producer.flush(1)
        time.sleep(2)
        self.consumer.close()
        self.producer.close()
    self.status="ERROR"
```

Engine podrá activar un botón KO con el que simulará fallos durante la ejecución del punto de carga con el fin de comprobar el correcto funcionamiento del sistema de notificación y la capacidad de respuesta de la central. Este botón también puede ser usado mientras el punto de carga está en uso, haciendo que el driver sea expulsado del punto y que el tiempo que ha estado conectado se mande a central.

```
<<MENU CHARGING POINT>>
Elige una de las opciones:
1. Enchufar coche
2. Desenchufar coche
3. Estado del engine
4. Volver al menu principal
```

Además, nuestro engine tendrá una interfaz por la cual puede suministrar sus recursos desde el propio punto. Esta tendrá la mismas funciones que tendrá el driver pudiendo conectar y desconectar el aparato del engine.

## d. Driver

El **Driver** es el cliente que realiza las peticiones de uso, las cuales son enviadas a la **Central** mediante **Kafka**.

Al ejecutar el programa, dentro de la función principal, se llama al **menú** que vemos a continuación:

```python
def menu():
    opciones = {"1": "AUTENTIFICACION", "2": "AUTORIZACION", "3": "ESTADO", "4": "FIN"}
    while True:
        print("\n===== MENÚ =====")
        print("1. AUTENTIFICACION")
        print("2. AUTORIZACION")
        print("3. ESTADO")
        print("4. FIN")
        eleccion = input("Selecciona (1-4): ").strip()
        if eleccion in opciones:
            return opciones[eleccion]
        print("ERROR: Opción no válida. Elige un número del 1 al 4.")
```

En este menú se ofrecen cuatro posibles opciones:

- **1. AUTENTIFICACIÓN:** Permite al driver identificarse ante la central para poder operar dentro del sistema.

- **2. AUTORIZACIÓN:** Solicita permiso para utilizar un punto de recarga específico.

- **3. ESTADO:** Consulta el estado actual de un punto de recarga, como su disponibilidad o nivel de actividad.

- **4. FIN:** Indica la finalización de una sesión o del proceso de recarga.

Cada opción devuelve una cadena con el tipo de petición seleccionada, que posteriormente será gestionada y enviada por Kafka hacia la central.

Las peticiones se envían a través de la función sendRequests(), cuya definición podemos ver a continuación:

```python
def sendRequests(bootstrap_server: str, driver_id: str):
    """
    Formato (4 campos): REM|PETICION|CP|DRIVER
      AUTENTIFICACION -> DRIVER|AUTENTIFICACION|NA|<DRIVER_ID>
      AUTORIZACION    -> DRIVER|AUTORIZACION|<CP_ID>|<DRIVER_ID>
      ESTADO          -> DRIVER|ESTADO|<CP_ID>|<DRIVER_ID>
      FIN             -> DRIVER|FIN|NA|<DRIVER_ID>
    """
    producer = KafkaProducer(
        bootstrap_servers=[bootstrap_server],
        value_serializer=lambda v: v.encode(FORMAT)
    )
    print(f"[DRIVER] Conectado a Kafka en {bootstrap_server}")

    while True:
        tipo = menu()

        # CP según el tipo
        if tipo in ("AUTORIZACION", "ESTADO"):
            cp_id = input("Introduce el ID del CP: ").strip()
            if not cp_id:
                print("ERROR: CP vacío. Cancelo envío.")
                continue
        else:
            cp_id = "None"

        msg = f"DRIVER|{tipo}|{cp_id}|{driver_id}"
        producer.send(TOPIC_DTC, msg)
        producer.flush()
        print(f"[DRIVER] Enviado → ({TOPIC_DTC}): {msg}")

        if tipo == "FIN":
            print("[DRIVER] Fin de sesión.")
            break

        time.sleep(DELAYS)

    producer.close()
```

En esta función se crea un productor Kafka que permite al Driver enviar mensajes a la Central.

Cada mensaje incluye cuatro campos: el remitente, el tipo de petición, el identificador del punto de recarga (CP) y el ID del driver.

Dentro de un bucle, la función llama al menú para que el usuario seleccione la operación que desea realizar.

Dependiendo del tipo de petición (AUTENTIFICACIÓN, AUTORIZACIÓN, ESTADO o FIN), se construye el mensaje con el formato correspondiente y se envía al topic destinado a la comunicación con la central.

Cuando el tipo seleccionado es FIN, el driver finaliza la sesión y se cierra la conexión Kafka correctamente.

Por supuesto, el Driver también puede recibir mensajes de respuesta enviados por la Central. Dichos mensajes se gestionan en la función receiveAnswers(), cuya definición podemos ver a continuación:

```
def receiveAnswers(bootstrap_server: str, driver_id: str):
    """
    Escucha respuestas de la Central en TOPIC_CTD.
    Formato esperado: RESPUESTA|<DRIVER_ID>|<MENSAJE>
    """
    consumer = KafkaConsumer(
        TOPIC_CTD,
        bootstrap_servers=[bootstrap_server],
        value_deserializer=lambda m: m.decode(FORMAT),
        auto_offset_reset="latest",
        enable_auto_commit=True,
        group_id="drivers-group",  # grupo común de drivers
    )
    print(f"[DRIVER] Escuchando respuestas en '{TOPIC_CTD}'…")
    for rec in consumer:
        text = rec.value or ""
        print(text)
        parts = text.split("|", 2)
        if len(parts) >= 3 and parts[1] == driver_id:
            mensaje = parts[2]
            print(f"[CENTRAL --> {driver_id}]: {mensaje}")

def main():
    print("****** EV_Driver ******")
    if len(sys.argv) < 3:
        print("Uso: python EV_Driver.py <BrokerIP:Puerto> <DriverID>")
        return

    bootstrap_server = sys.argv[1]
    driver_id = sys.argv[2]

    t = threading.Thread(target=receiveAnswers, args=(bootstrap_server, driver_id), daemon=True)
    t.start()

    sendRequests(bootstrap_server, driver_id)
    print("Cerrando Driver…")
    time.sleep(1)
```

En esta función se crea un consumer Kafka que permanece a la escucha de los mensajes publicados por la central en el topic correspondiente.

Cada vez que llega una respuesta, se muestra su contenido en pantalla, indicando el mensaje recibido y el topic desde el que proviene.

El bucle de escucha se mantiene activo mientras la comunicación esté abierta, permitiendo al driver recibir en tiempo real las confirmaciones o actualizaciones procedentes de la central.

Cuando el proceso finaliza o se interrumpe la conexión, el consumer se cierra de forma controlada.

# 3. Despliegue

**Arrancar la Central:**
Primero se debe ejecutar la **Central**, pasándole como parámetro la **IP del ordenador** donde se está ejecutando.
Esta IP será utilizada por el resto de componentes (monitor, engine y driver) para establecer la comunicación con la central.

**Arrancar el Monitor y el Engine:**
A continuación, se inician el **Monitor** y el **Engine** en **otro ordenador** que comparta la **misma red Ethernet** que el equipo donde se encuentra la Central.
Estos dos componentes se ejecutarán desde el archivo "ChargingPoint.py".
Al archivo se les debe pasar los **parámetros necesarios** (--central <IP de central:Puerto de central> --engine <ip de engine:puerto de engine>, --id <id del charging point> --localizacion <localizacion del CP> --broker <IP del broker:Puerto del broker>) para que puedan conectarse correctamente y empezar a intercambiar mensajes.

**Arrancar el Driver:**
Finalmente, se ejecuta el **Driver** desde un **tercer ordenador**, también conectado a la misma red local.
Este componente será el encargado de enviar las peticiones a la Central y recibir las respuestas a través de Kafka.