4-11-2025

PRÁCTICA 1

PRÁCTICA EVCHARGING
SISTEMAS DISTRIBUIDOS

Adrián Requena Fernández 24509047V
Ismael Boudot Martínez 54207260D
GRADO EN INGENIERÍA INFORMÁTICA

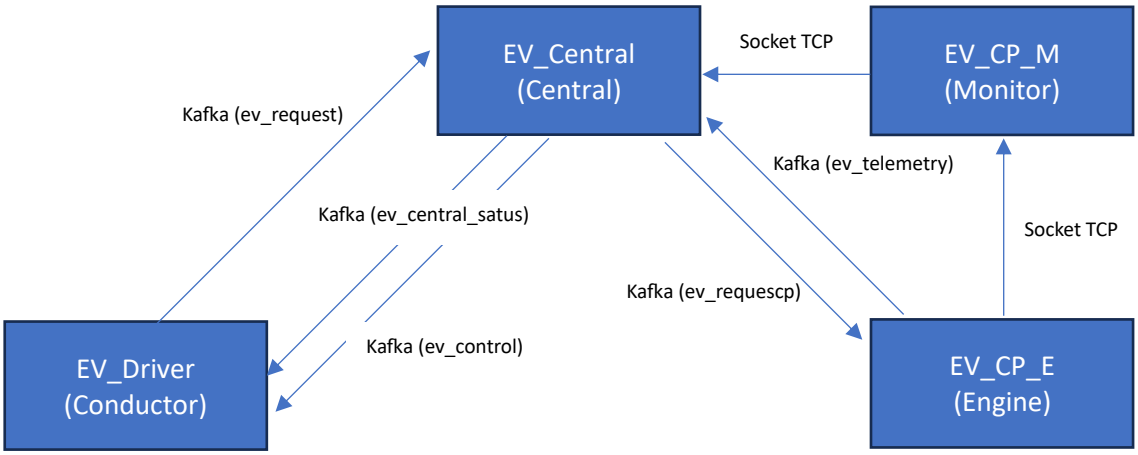
ÍNDICE

1. INTRODUCCIÓN

Esta práctica implementa un sistema distribuido para la gestión de una red de puntos de recarga de vehículos eléctricos. El sistema simula una solución completa que incluye monitorización centralizada, puntos de recarga autónomos y aplicación para conductores, utilizando tecnologías como sockets TCP y Apache Kafka para la comunicación entre componentes.

2. ARQUITECTURA DEL SISTEMA

2.1. Diagrama de componentes



2.2. Descripción de los módulos

- EV_Central: Sistema central de monitorización y control. Gestiona el estado de toda la red, recibe solicitudes de conductores y coordina con los puntos de recarga.
- EV_CP_E: Motor del punto de recarga. Simula el proceso de carga y se comunica con la central para reportar telemetría.
- EV_CP_M: Monitor de salud del punto de recarga. Supervisa el estado del motor y reporta a la central.
- EV_Driver: Aplicación del conductor. Permite solicitar servicios de recarga y monitorizar el progreso.
- Apache Kafka: Sistema de mensajería para comunicación asíncrona entre componentes.

3. PROTOCOLOS DE COMUNICACIÓN

3.1. Comunicación vía sockets

Mensajes entre EV_CP_M y EV_Central:

- CONNECTION#<CP_ID> - Registro inicial
- ALTA#<CP_ID>#<ubicación>#<precio> - Registro dinámico de un nuevo CP
- KEEPALIVE#<CP_ID> - Latido de conectividad
- ERROR#<CP_ID> - Reporte de avería

Mensajes entre EV_CP_M y EV_CP_E:

- CONNECTION#<CP_ID> - Asignación de identificador

3.2. Comunicación vía Kafka

Topics configurados:

- ev_request - Solicitudes de conductores -> conductor a central
- ev_requestcp - Solicitudes de central -> puntos de carga
- ev_telemetry - Datos de telemetría -> central
- ev_control - Mensajes de control -> conductores
- ev_central_status – Monitorear la salud de la central a los otros módulos -> conductores, monitor

Mensajes clave

- REQUEST#<cp_info>#<kwh_solicitados>#<id_conductor> - Solicitud de recarga
- CHARGING#<cp_id>#<driver_id>#<kwh_actual>#<kwh_solicitados> - Progreso de carga
- END#<cp_id>#<driver_id>#<mensaje_precio> - Finalización de carga
- CENTRAL_STATUS#ALIVE - Latido de la central
- RELOAD - Solicitud de actualización de lista de CPs
- RELOAD#<lista_cps> - Respuesta con lista actualizada

4. COMPONENTES SOFTWARE DESARROLLADOS

4.1. EV_Central

4.1.1. Descripción funcional

El módulo EV_Central es el núcleo del sistema de gestión de la red de carga. Implementa las siguientes funcionalidades principales:

- Gestión de puntos de carga: Mantiene un registro de todos los CPs con sus estados (CONECTADO, DESCONECTADO, CARGANDO, AVERIADO, PARADO).
- Procesamiento de solicitudes: Maneja las peticiones de carga provenientes de los conductores (EV_Driver).
- Monitorización en tiempo real de toda la red: Proporciona una interfaz gráfica para visualizar el estado de toda la red.
- Comunicación asíncrona: Utiliza Kafka para la comunicación con otros componentes del sistema.
- Gestión de heartbeats: Verifica la conectividad de los CPs mediante un sistema de latidos.
- Almacenamiento de datos: Almacena la información de los CPs en un archivo de texto, que se encuentra en la carpeta raíz del proyecto con el nombre cpdatabase.txt.

4.1.2. Parámetros de configuración

El módulo requiere los siguientes parámetros de línea de comandos:

java -jar EV_Central.jar <puerto_socket_CP> <ip_broker_kafka>

Ejemplo:

Java -jar EV_Central.jar 8888 hostIP:9092

En la carpeta de Kafka (el archivo .env) detecta la ip del pc en el que se ha iniciado kafka automáticamente. También es posible comprobarlo manualmente en la consola de mandos.

4.1.3. Estructura del código

Clases Principales:

EV_Central (clase main)

- Función: Punto de entrada principal del sistema.
- Responsabilidades:
  - Inicialización de la interfaz gráfica.
  - Carga inicial de CPs desde la base de datos.
  - Inicio de consumidores Kafka para diferentes topics.
  - Inicio del servidor socket para comunicación con CPs.

CentralLogic

- Función: Lógica principal.
- Métodos Clave:
  - handleRequest(): Procesa solicitudes de carga.
  - handleTelemetry(): Maneja datos de telemetría.
  - handleCP(): Gestiona los mensajes de los puntos de carga.
  - Gestión del mapa de heartbeats.

CentralMonitorGUI

- Función: Interfaz gráfica de monitorización.
- Características:
  - Visualización de CPs con colores según estado.
  - Panel de solicitudes en curso.
  - Panel de mensajes del sistema.
  - Botones de control para parar/reanudar CPs.

HeartbeatChecker

- Función: Monitorización de conectividad de CPs.
- Funcionalidad: Verifica periodicamente los últimos heartbeats recibidos.

4.1.4. Flujos de comunicación

Con EV_Driver (vía Kafka):

- Topic: REQUEST - Solicitudes de carga.
- Topic: TELEMETRY - Datos de telemetría.

Con EV_CP (vía Socket):

- Protocolo: Comunicación directa TCP mediante sockets.
- Mensajes: CONNECTION, KEEPALIVE, ERROR.

Con sistema de monitorización:

- Actualización en tiempo real de la interfaz gráfica cada segundo.

4.1.5. Estados gestionados

DESCONECTADO // Color GRIS - No conectado al sistema

CONECTADO // Color VERDE - Disponible para carga

CARGANDO // Color VERDE - Suministrando energía

AVERIADO // Color ROJO - Con avería

PARADO // Color NARANJA - Fuera de servicio por orden central

4.2. EV_CP_E

4.2.1 Descripción funcional

El módulo EV_CP_E representa el motor principal del punto de recarga. Sus responsabilidades incluyen:

- Gestión de solicitudes de carga: Procesa las peticiones de recarga recibidas a través de Kafka desde la central.
- Simulación de proceso de carga: Implementa el flujo completo de suministro de energía.
- Comunicación con monitor vía socket: Recibe el identificador del CP desde el módulo monitor.
- Reporte de Telemetría: Envía actualizaciones periódicas del estado de carga a la central.

4.2.2. Parámetros de configuración

El módulo requiere los siguientes parámetros:

java -jar EV_CP_E.jar <ip_broker_kafka> <puerto_socket_CP>

Ejemplo:

java EV_CP_E.jar ipPCKafka:9092 9999

4.2.3. Estructura y flujo principal

Inicialización:

- Establece servidor socket para comunicación con el monitor.
- Inicia consumidor Kafka para recibir solicitudes de carga.

Procesamiento de solicitudes:

```
static void handleDriverRequest(String message) {
 // Formato: REQUEST#ALC001;0.35;Carrer de Sant Vicent;CONECTADO#10.0#DV001#0.35
        if(!CP.equals(message.split("#")[1].split(";")[0])) return;

        Producer r = new Producer(broker, CommonConstants.TELEMETRY);
        r.sendMessage("ACKREQUEST#" + CP + "#" + message.split("#")[3]);
```

Simulación de carga:

El motor simula el proceso de carga mediante un bucle que:

- Incrementa progresivamente los kWh suministrados.
- Envía actualizaciones cada segundo a la central.
- Finaliza cuando se alcanza la cantidad solicitada.

4.2.4. Protocolos de comunicación

Con EV_CP_M (Socket):

- Mensaje: CONNECTION#<CP_ID> - Asignación de identificador.

Con EV_Central (Kafka):

- Topic: TELEMETRY - Envío de estados de carga.
- Topic: REQUEST_CP - Recepción de solicitudes.

Tipos de mensajes enviados:

- ACKREQUEST: Confirmación de recepción de solicitud.
- CHARGING: Actualización de progreso de carga.
- END: Finalización del proceso de carga.

4.2.5. Características de implementación

- Comunicación asíncrona: Uso de Kafka para no bloquear el proceso principal.
- Simulación en tiempo real: Pausas de 1 segundo entre actualizaciones.
- Validación de identidad: Verificación de que la solicitud es para el CP correcto.
- Gestión de recursos: Cierre adecuado de productores Kafka.
- Simulación de caída del Engine al escribir 'K' y pulsar enter en la consola de comandos.

4.3. EV_CP_M

4.3.1. Descripción funcional

El módulo EV_CP_M actúa como el sistema de monitorización y salud del punto de recarga. Sus principales responsabilidades son:

- Monitorización continua: Verifica el estado del CP (EV_CP_E) mediante conexiones periódicas.
- Gestión de heartbeats: Envía latidos regulares a la central para confirmar conectividad.
- Detección de fallos: Identifica desconexiones o malfuncionamientos del engine.
- Comunicación bidireccional: Mantiene conexiones simultáneas con el engine y la central.
- Interfaz: Proporciona una interfaz para visualizar el estado local del CP y de la central.

4.3.2. Parámetros de configuración

El módulo requiere los siguientes parámetros:

java -jar EV_CP_M.jar <ip_engine:puerto> <ip_central:puerto> <id_cp>

Ejemplo:

java -jar EV_CP_M.jar localhost:9091 localhost:9090 ALC001

Si quieres dar de alta un nuevo CP el comando será:

java -jar EV_CP_M.jar <ip_engine:puerto> <ip_central:puerto> <id_cp> <ubicación> <precio>

4.3.3. Estructura del componente

EV_CP_M (clase main)

- Función: Coordinador principal del monitor.
- Responsabilidades:
  - Inicialización de la interfaz gráfica.
  - Creación de hilos para monitorización y heartbeats.
  - Gestión de cambios de estado del engine.

ConnectionToEngine

- Función: Monitorización continua del engine del CP.
- Características:
  - Intenta conexión cada 1 segundos.
  - Notifica a la central sobre cambios de estado.
  - Reconexión automática en caso de fallos.

MonitorKeepAlive

- Función: Envío periódico de heartbeats a la central.
- Frecuencia: 1 latido por segundo.
- Mantiene a la central informada sobre la conectividad.

MonitorGUI

- Función: Interfaz local de monitorización.
- Capacidades: Muestra hasta 25 mensajes de estado en tiempo real.

4.3.4. Flujos de comunicación clave

Monitorización del engine:

```
while (isConnected && !Thread.currentThread().isInterrupted()) {
   String statusRequest = "CONNECTION#" + ID_CP;
   String response = cpClient.sendMessage(statusRequest);
   Thread.sleep(5000); // Verificación cada 5 segundos
}
```

Heartbeats a la central:

```
while (!Thread.currentThread().isInterrupted()) {
   String statusRequest = "KEEPALIVE#" + cpID;
   cpClient.sendOnly(statusRequest);
   TimeUnit.SECONDS.sleep(HEARTBEAT_INTERVAL_SECONDS); // 1 segundo
}
```

Gestión de estados:

- Engine Conectado: EngineAlive() -> Envía CONNECTION#<CP_ID> a central.
- Engine Desconectado: LostEngineConnection() -> Envía ERROR#<CP_ID> a central.

4.3.5. Protocolos de mensajería

Hacia EV_Central (Socket):

- CONNECTION#<CP_ID> - Registro/Reconexión exitosa
- KEEPALIVE#<CP_ID> - Latido de conectividad
- ERROR#<CP_ID> - Reporte de malfuncionamiento

Hacia EV_CP_E (Socket):

- CONNECTION#<CP_ID> - Solicitud de estado del engine

4.3.6. Mecanismos de resilencia

- Reintentos automáticos: Reconexión automática cada 10 segundos en caso de fallo.
- Monitorización independiente: El monitor funciona incluso si el engine falla.
- Buffer de mensajes: Limitación a 25 mensajes para evitar desbordamiento de memoria.
- Comunicación separada: Uso de hilos separados para diferentes funciones.

4.4. EV_Driver

4.4.1. Descripción funcional

El módulo EV_Driver es la aplicación utilizada por los conductores para interactuar con la red de carga. Sus principales funcionalidades son:

- Interfaz de usuario: Proporciona una interfaz intuitiva para solicitar servicios de carga.
- Gestión de solicitudes: Permite solicitar carga tanto manualmente como mediante scripts automatizados.
- Monitorización en tiempo Real: Muestra el progreso de las cargas activas.
- Comunicación Asíncrona: Se integra con el sistema central mediante Kafka.

4.4.2. Parámetros de configuración

El módulo requiere los siguientes parámetros:

java -jar EV_Driver.jar <ip_broker:puerto> <id_conductor>

Ejemplo:

java -jar EV_Driver.jar ipPCKafka:9092 DV001

4.4.3. Estructura del componente

EV_Driver (Clase Main)

- Función: Punto de entrada de la aplicación del conductor.
- Responsabilidades:
  - Inicialización de la interfaz gráfica.
  - Configuración del productor/consumidor Kafka.
  - Gestión de la comunicación con el sistema central.

DriverGUI

- Función: Interfaz gráfica del conductor
- Características:
  - Panel de puntos de recarga que existen en la base de datos.
  - Botones para solicitar carga manual.
  - Ejecución de scripts automatizados.
  - Actualizar los Cps disponilbles en la base de datos
  - Mensajes de logs en tiempo real.

4.4.4. Flujos de operación

Solicitud manual de carga

```
String mensajeKafka = "REQUEST#" + linea + "#10.0#" + name;
producer.sendMessage(mensajeKafka);
Log("Solicitando Recarga...");
Suministrando = true;
```

Ejecución de script automatizado:

```
File archivo = new File("request_script.txt");
Scanner s = new Scanner(archivo);
while (s.hasNextLine()) {
   if(Suministrando) continue;
   String linea = s.nextLine();
   // Procesamiento de cada línea del script
}
```

Formato del script de carga:

```
REQUEST#ALC001#10.0
REQUEST#ALC002#15.5
REQUEST#MAD001#8.0
…
```

4.4.5. Protocolos de comunicación

Hacia EV_Central (Kafka):

- Topic: REQUEST - Solicitudes de carga
- Formato: REQUEST#<cp_info>#<kwh_solicitados>#<id_conductor>

Desde EV_Central (Kafka):

- Topic: CONTROL - Respuestas y actualizaciones
- Tipos de Mensajes:
  - CHARGING - Progreso de carga
  - END - Finalización de carga
  - NOAVIABLE - CP no disponible

4.4.6. Mecanismos de control

Gestión de estado:

public boolean Suministrando = false;

Este flag evita solicitudes simultáneas y garantiza que el conductor solo tenga una carga activa a la vez.

Buffer de logs:

```
if(logBuffer.size() > 8) {
        logBuffer.remove(0);
}
```

Mantiene un historial limitado de mensajes para una visualización clara.

Filtrado de mensajes:

if(!message.contains(ID_Driver)) return;

Garantiza que cada conductor solo reciba mensajes destinados a él.

4.4.7. Características de la interfaz

- Diseño automático: Grid layout que se adapta al número de CPs.
- Respuesta inmediata: Logs en tiempo real de todas las operaciones.
- Prevención de duplicados: Bloqueo de múltiples solicitudes simultáneas.
- Integración con base de datos: Carga automática de CPs desde archivo cpdatbase.txt.
- Actualización de CPs: Actualiza la lista de CPs disponibles en la base de datos de la Central.

4.5. EVCharging-Common (clases comunes)

4.5.1. Clase CP

Almacena la información de un punto de recarga. Incluye un método toString() para serialización.

```
public class CP {
        public String UID;
        public String Price;
        public String Location;
        public String State;
        public float KWHRequested = 0.f;
        public String driver = null;
```

4.5.2. Clase CommonConstants

Contiene los topics de Kafka utilizados en el sistema.

```
        public static String REQUEST = "ev_request";
        public static String REQUEST_CP = "ev_requestcp";
        public static String TELEMETRY = "ev_telemetry";
        public static String CONTROL = "ev_control";
        public static String CENTRAL_STATUS = "ev_central_status";
```

4.5.3. Clases de Comunicación

- EVClient: Cliente TCP para comunicación por sockets.
- EVServerSocket: Servidor TCP multihilo.
- EVClientHandler: Manejador de conexiones cliente.
- Producer: Productor Kafka para envío de mensajes.
- KConsumer: Consumidor Kafka para recepción de mensajes.

5. GUÍA DE DESPLIEGUE

5.1. Requisitos del sistema

- Java 8 o superior, o JRE
- Docker y Docker Compose
- El sistema esta desarrollado en Windows, no ha sido probado en más sistemas operativos.

5.2. Configuración con Docker

Archivo docker-compose.yml:

```
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.6.1
    ports: ["2181:2181"]

  kafka:
    image: confluentinc/cp-kafka:7.6.1
    ports: ["9092:9092"]
    environment:
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://${HOST_IP}:9092

  kafka-topics-creator:
    image: confluentinc/cp-kafka:7.6.1
    depends_on: [kafka]
    command: [...]
```

Desplegar Kafka

Para desplegar Kafka simplemente hay que entrar en la carpeta de proyecto Despliegue/Kakfa darle doble clic en el archivo start_kafka.bat y en unos 15 segundos Kafka estará iniciado automáticamente.

- Detecta automáticamente la IP del host (la guarda en el archivo .env) Si la IP que coge automáticamente es incorrecta, modificar en el archivo Docker-compose.yml la línea KAFKA_ADVERTISED_LISTENERS cambiando $Host_IP por la IP real.
- Modifica la ip de Kafka
- Inicia los topics de Kafka

5.3. Pasos de despliegue

En la carpeta del proyecto se incluye una carpeta llamada Despliegue con la que se ejecutarán todos los componentes necesarios. Hay que llevar cuidado a la hora de editar los archivos de configuración y no dejas espacios en blanco, tanto al principio como al final. Los pasos para seguir son:

1. Iniciar Kafka

Desde la capeta Kafka ejecutamos el archivo:

start-kafka.bat

Si la IP que coge automáticamente es incorrecta, modificar en el archivo Docker-compose.yml la línea KAFKA_ADVERTISED_LISTENERS cambiando $Host_IP por la IP real.

2. Desplegar EV_Central

Desde la carpeta Central ejecutamos el archivo (modificar los argumentos del archivo args.txt antes de ejecutar):

run_central.bat

3. Desplegar puntos de carga

Hay 3 opciones de despliegue. Desde la carpeta CP ejecutamos:

- *run_cp.bat (despliega un único cp con engine y monitor, modificar* los argumentos del archivo config_cp.txt antes de ejecutar)
- *run_all_cps.bat (despliega todos los cps que hay indicados en el archivo, hay que modificar los argumentos del .bat)*
- *alta_cp.bat (agrega un nuevo cp a la base de datos y lo despliega, modificar el archivo alta_cp_config.txt antes de ejecutar)*

4. Desplegar aplicaciones de conductor

Hay 2 opciones de despliegue. Desde la carpeta Driver ejecutamos:

- *run_driver.bat (despliega un único driver, modificar el archivo config_driver.txt)*
- *run_drivers.bat (despliega n drivers, modificar el archivo config_drivers.txt)*

5.4. Escenario de despliegue distribuido

PC1 (Conductores):

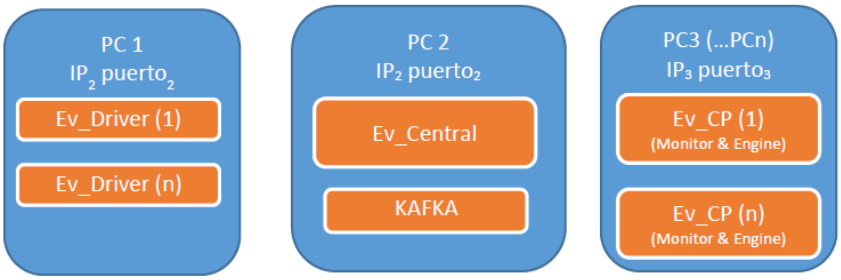
- Múltiples instancias de EV_Driver

PC2 (Central):

- EV_Central
- Kafka

PC3 (Puntos de Carga):

- Múltiples instancias de EV_CP_E y EV_CP_M



6. MANUAL DE USUARIO

6.1. Flujo de operación normal

1. Inicio del Sistema:

- Ejecutar EV_Central primero.
- Ejecutar puntos de recarga (EV_CP_M + EV_CP_E).
- Ejecutar aplicaciones de Driver.

2. Solicitud de Recarga:

- Manual: Click en botón del CP deseado.
- Automática: Ejecutar script de suministro.

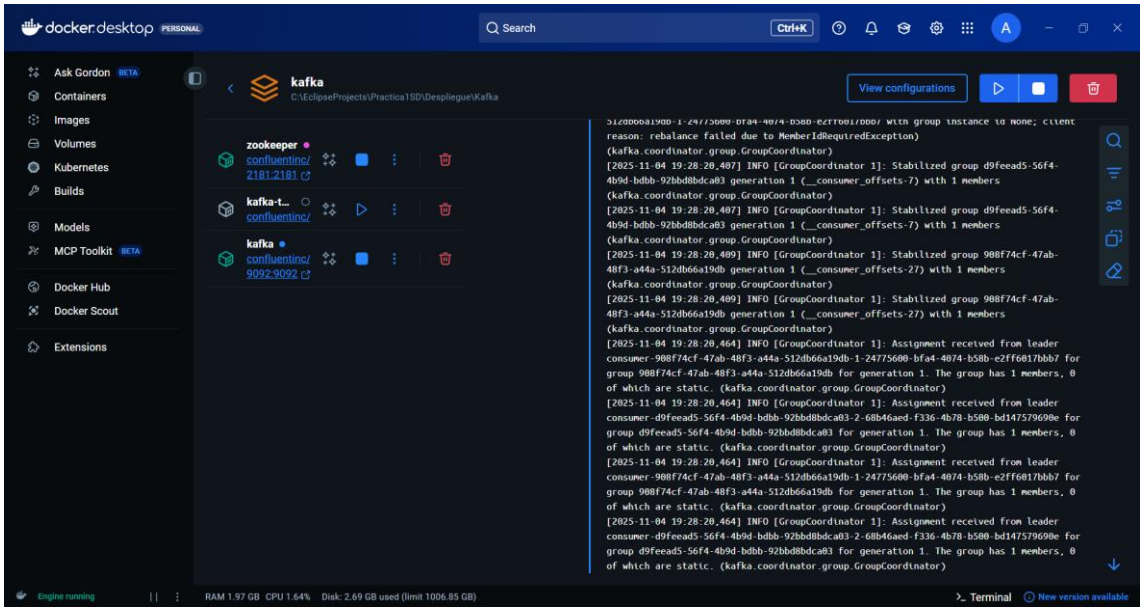
3. Monitorización:

- Central: Visualiza estado de todos los CPs.
- Driver: Sigue progreso de recarga actual.
- Monitor: Verifica salud del punto de recarga.

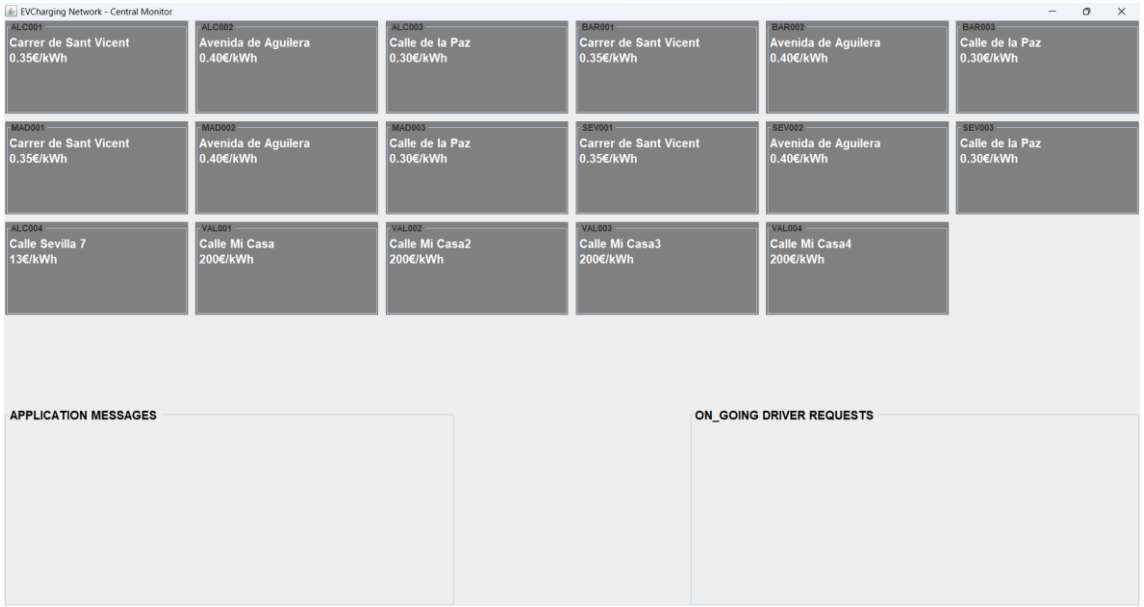
6.2. Gestión de incidencias

- Avería durante recarga: Finalización inmediata y notificación.
- Desconexión de CP: Cambio automático a estado "Desconectado".
- Reconexión: Restauración automática del estado anterior.

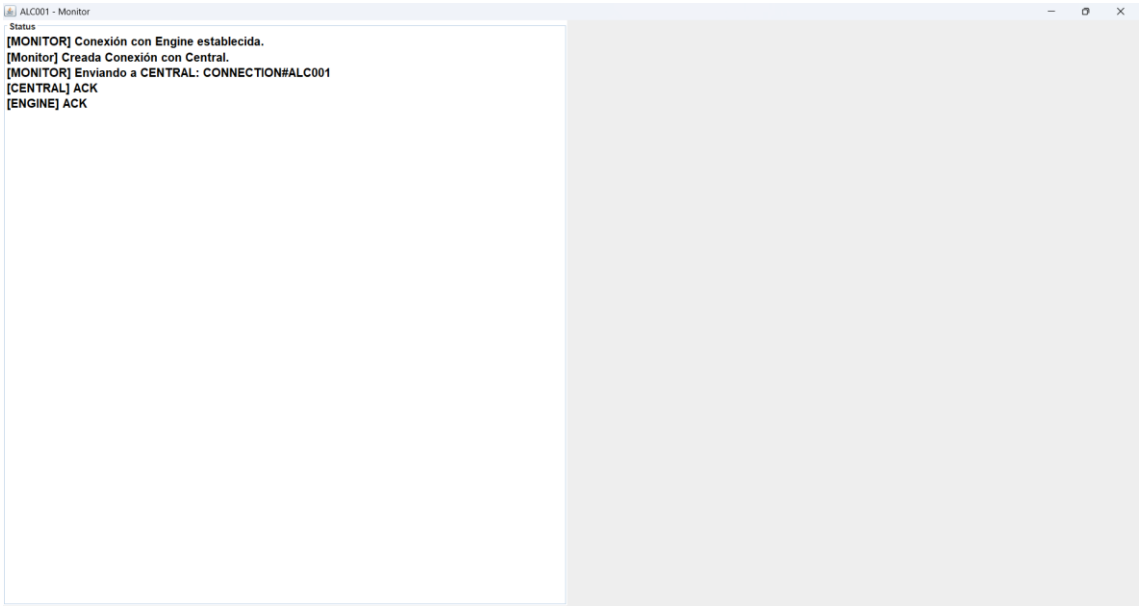
7. CAPTURAS DE PANTALLA



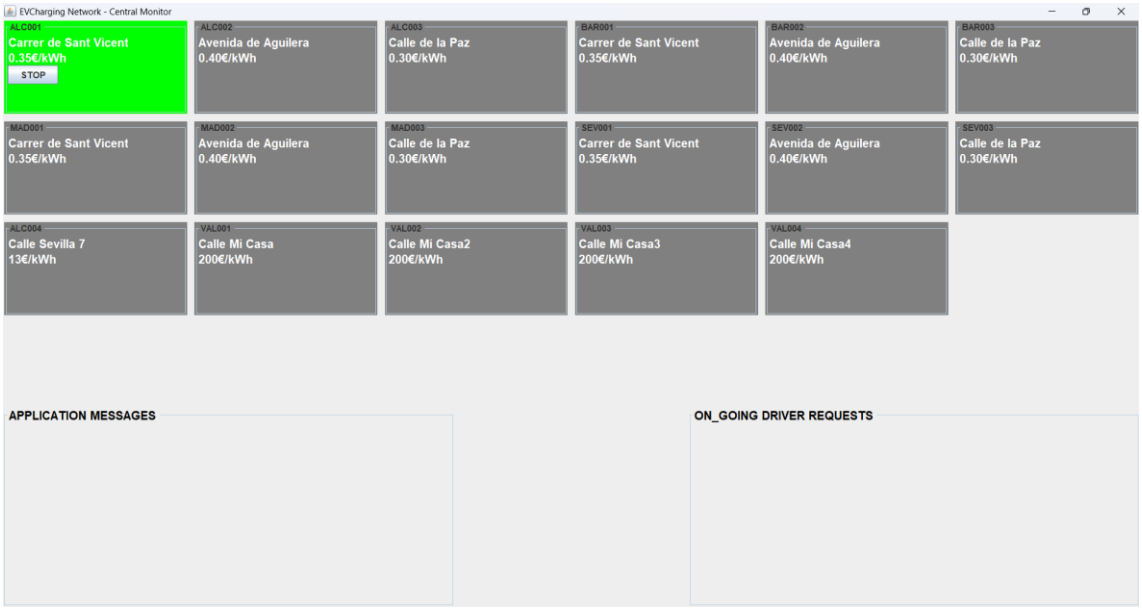
Kafka iniciado.



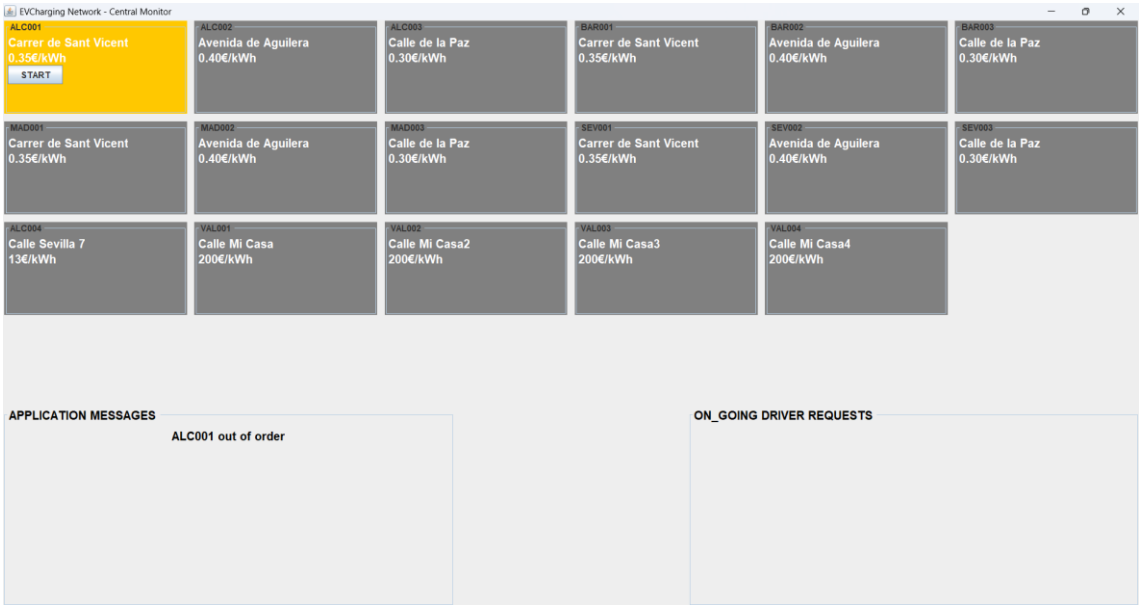
Central desplegada y esperando CPs.



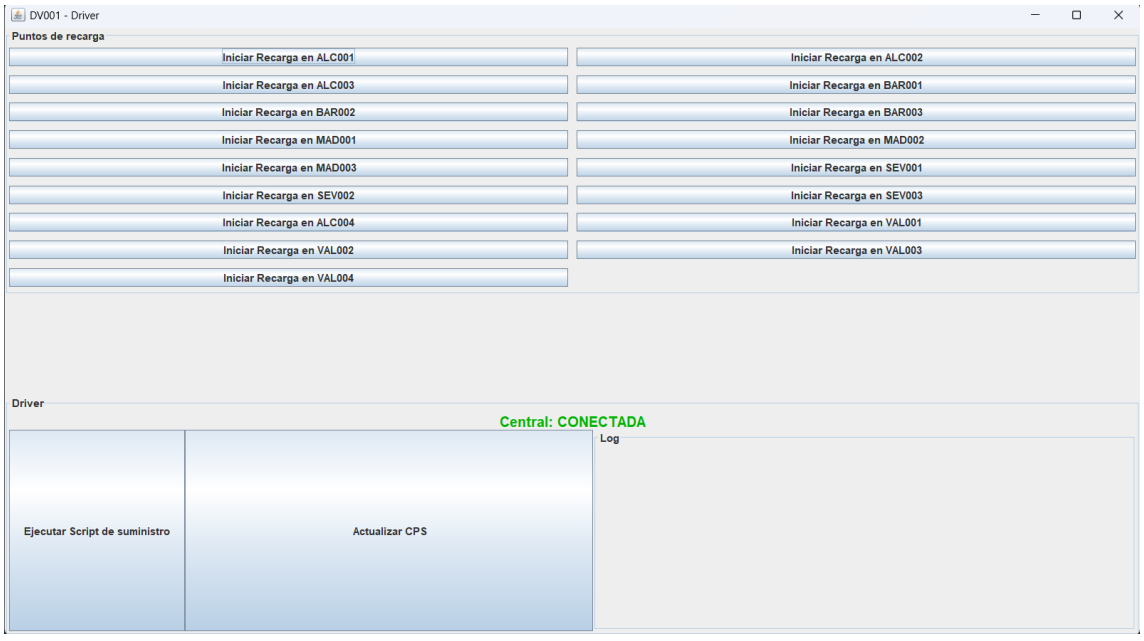
Engine y Monitor desplegados y conectados a Central.



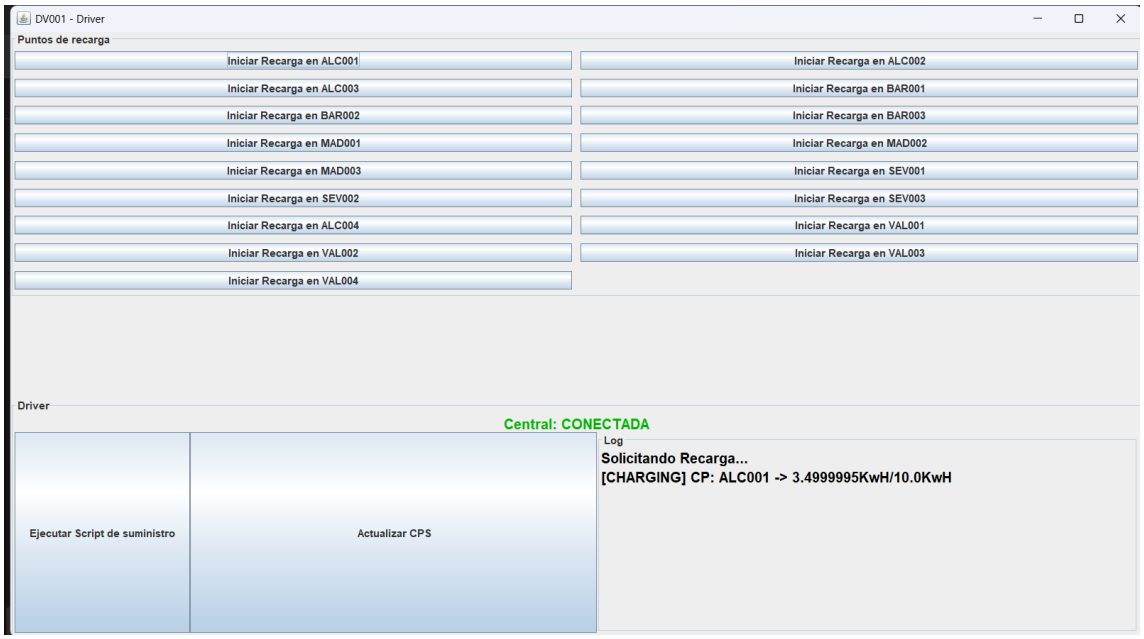
CP conectado y preparado para suministrar carga.



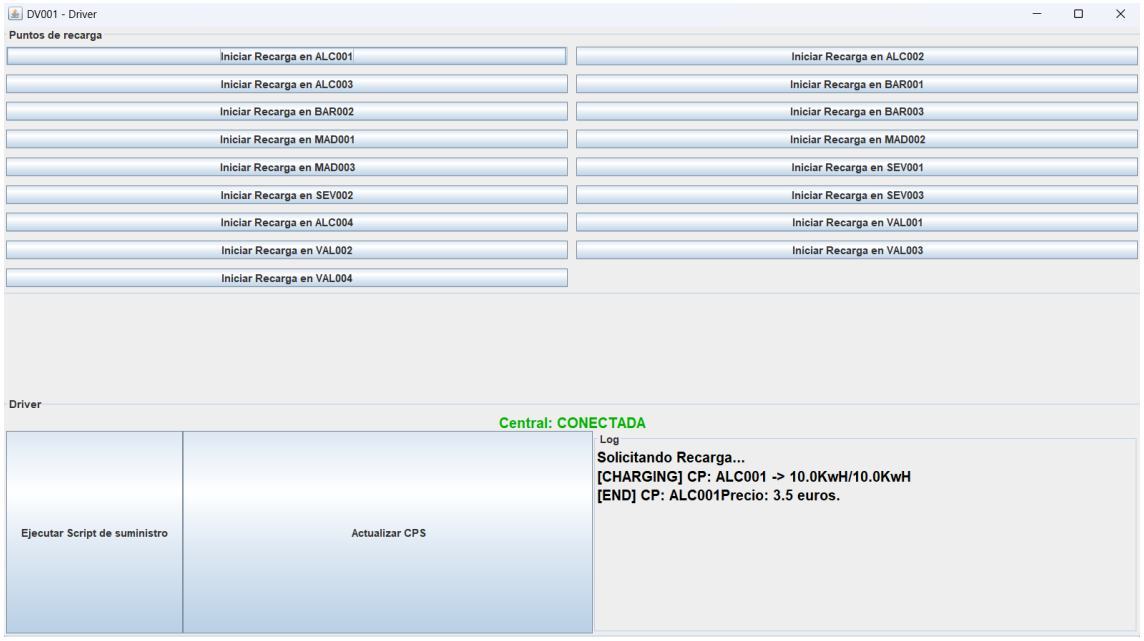
Desde Central podemos manualmente poner en Parado un CP.



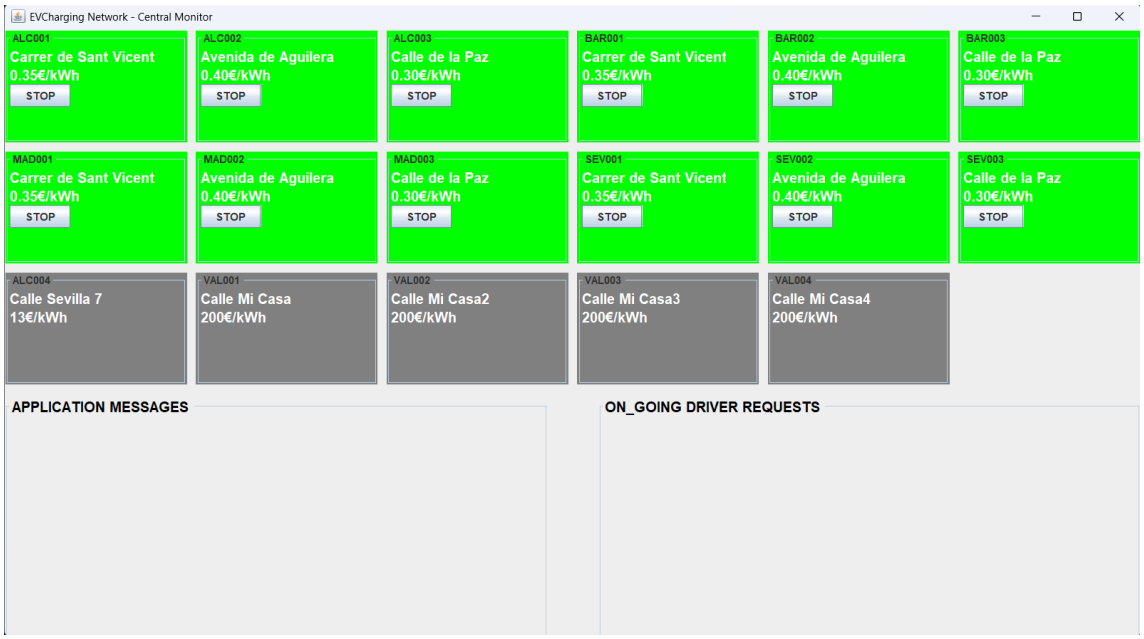
Driver desplegado correctamente y conectado a Central.



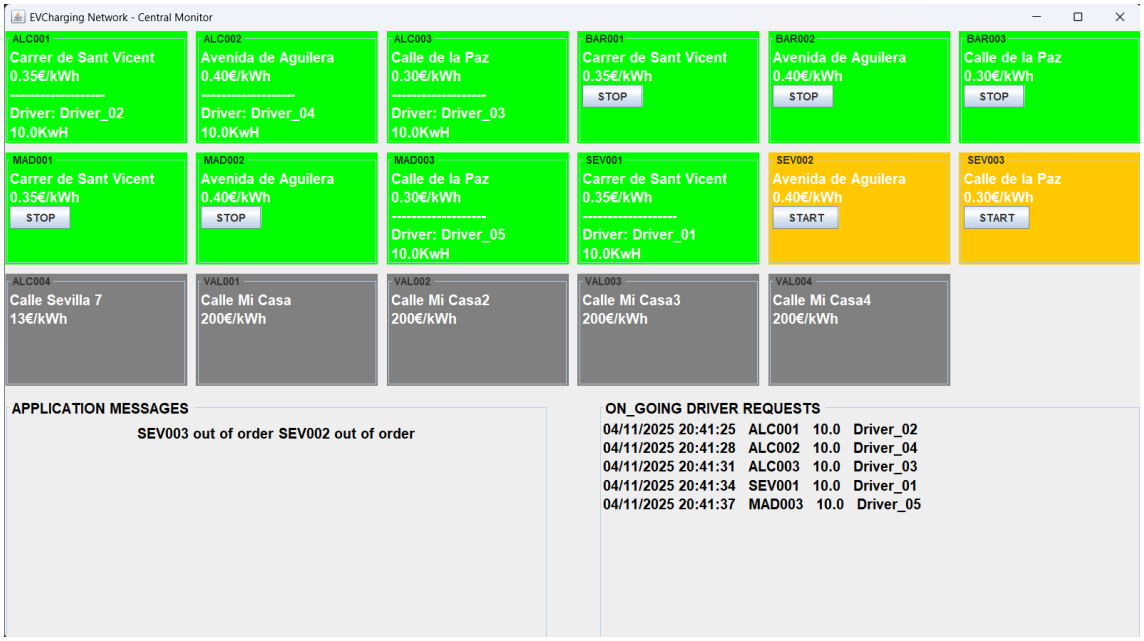
Driver suministrando del CP seleccionado



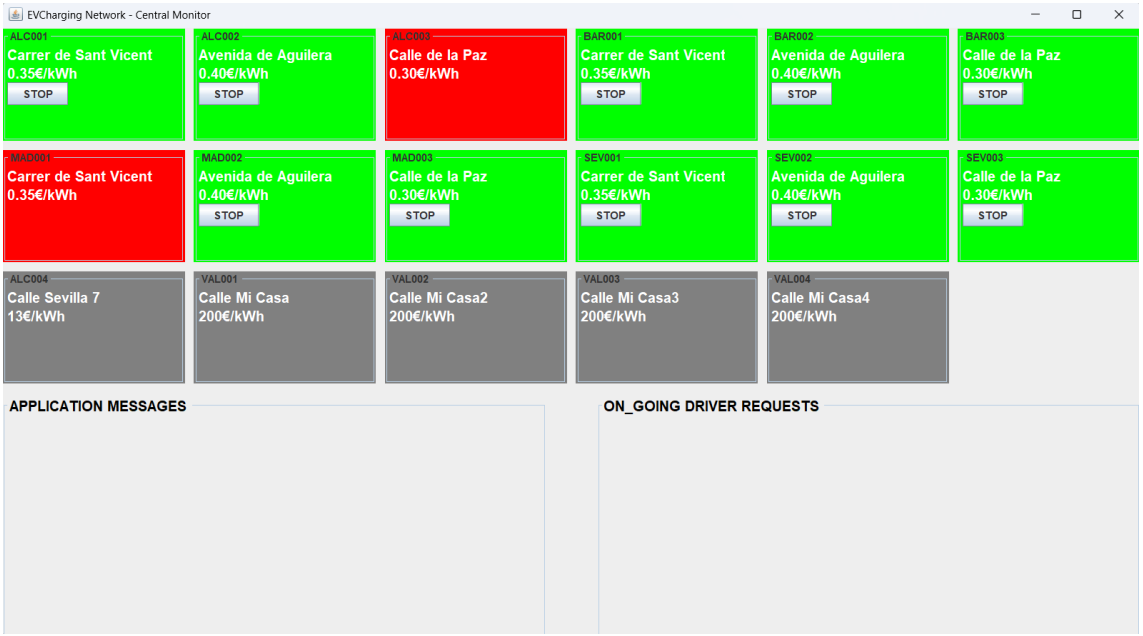
Suministro completado.



Varios CPs iniciados y conectados a Central.



Funcionamiento de varios Drivers solicitando suministro a varios Cps distintos. (funcionamiento real del sistema).



CPs en rojo averiados.

8. CONCLUSIONES

8.1. Dificultades encontradas

- Sincronización entre componentes distribuidos.
- Gestión de estados concurrentes.
- Configuración de red para despliegue distribuido.
- Manejo de desconexiones de los componentes.

8.2. Soluciones implementadas

- Sistema de heartbeats para detección de caídas.
- Mecanismos de reconexión automática.
- Comunicación asíncrona con Kafka.

8.3. Aprendizajes obtenidos

- Arquitectura de sistemas distribuidos.
- Patrones de comunicación (sockets, topics).
- Gestión de concurrencia y estados.
- Aprender a desplegar usando contenedores en Docker.