Distributed Systems

Sergio Pascual Morcillo 20097175M

Nicolás Idelfonso Sirvent Orts DNI

Universitat d'Alacant
Universidad de Alicante

EV Charging Network

2025 - 2026

Index

Software components

EV_Central

/data

- Charging_Point.db

Database where

CChargingPoint

The file defines the `CChargingPoint` class, which models an **electric vehicle charging point** within a charging management system. Each object represents a charger with attributes such as identifier, location, price per kW, operational status, energy consumption, and total cost.

The class imports `sqlite3` to interact with a database and `CPStatus` to manage the different operational states of the charging point. Its main methods include:

- **`__init__`**: initializes the charger's attributes.
- **`__eq__`**: allows comparison between two charging point objects.
- **`turn_ON`**: intended to activate the charging point (currently not implemented).
- **`turn_OFF`**: stops the charging process, resets its values, and updates the status in the SQLite database.

central_config

The `CentralConfig` module defines a configuration handler for the **central system** of a charging infrastructure or distributed service. Its purpose is to manage runtime parameters—such as network addresses and ports—using command-line arguments.

The class **`CentralConfig`** automatically retrieves configuration values at initialization through the private static method `_get_central_config()`, which employs Python's `argparse` library. This design ensures flexible deployment, allowing parameters like the central server's IP, port, and Kafka connection details to be specified dynamically when the program starts.

Key elements:

- **`__init__`**: initializes configuration attributes (`ip`, `port`, `kafka_ip`, `kafka_port`) based on parsed arguments.

- **`_get_central_config()`**: defines and processes command-line arguments, returning them as a dictionary.

Overall, the module encapsulates **configuration management** in a structured and reusable way, promoting scalability and portability across different environments.

- compose.yaml

This file defines the **containerized environment** for the system using Docker Compose. It specifies three core services—**Central**, **Kafka**, and **Database**—which together form the software infrastructure required for the charging management system.

cp_status

The `cp_status.py` module defines the **status management system** for a charging point, encapsulating its operational states through enumerations and helper methods.

The `CPStatusOptions` enumeration lists all possible states a charging point can assume—such as *ACTIVE*, *SUPPLYING*, *STOPPED*, *WAITING_FOR_SUPPLY*, *BROKEN_DOWN*, and *DISCONNECTED*—each represented by a unique integer value.

The `CPStatus` class provides a structured interface to manipulate and query these states. It initializes each instance in the *DISCONNECTED* state and includes setter methods (e.g., `set_active()`, `set_supplying()`) to update the current status, as well as boolean checkers (e.g., `is_active()`, `is_stopped()`) to verify the system's condition.

directivers_producer

The `DirectivesProducer` module implements a **Kafka-based communication component** responsible for sending control directives from the central system to charging points or other subsystems.

The class uses the `confluent_kafka` library to produce messages to the Kafka topic **`central-directives`**, ensuring reliable and scalable message delivery. Each directive is formatted as a JSON object containing a target identifier, an action (e.g., *start*, *stop*, *resume*), and optionally a supply session ID.

Key methods include:

- **`start_supply()`**, **`stop_cp()`**, **`resume_cp()`** – send specific commands to individual charging points.

- **`stop_all()`**, **`resume_all()`** – broadcast directives to all connected devices.

Internally, the `_send_directive()` method handles message construction and transmission.

dockerfile

The `Dockerfile` defines the **containerized runtime environment** for the central system component. It builds upon the lightweight `python:3.12-slim` base image to ensure efficiency and compatibility.

Key instructions include:

- **`RUN apt update && apt -y install tk`** – installs the Tkinter library, often required for graphical or system-level dependencies.

- **`RUN pip install confluent-kafka`** – installs the Kafka client library used for message production and consumption.

- **`WORKDIR /usr/src/EV_Central`** – sets the working directory inside the container.

- **`COPY . .`** – copies all project files into the container's workspace.

- **`ENTRYPOINT [ "python", "main.py" ]`** – defines the command to execute when the container starts, launching the main application script.

GUI

The `GUI` module implements a **graphical monitoring and control interface** for the electric vehicle charging management system, built using Python's `tkinter` library. Its primary purpose is to visualize the operational status of all charging points, manage system messages, and allow the operator to control charging activity interactively.

The `CentralApp` class initializes a user interface that displays each charging point as a colored panel, where the color reflects its current status (active, supplying, stopped, etc.), as defined in the `STATES` dictionary. Each panel includes basic information—ID, location, and price—and a button to toggle power states (`Power_ON` / `Power_OFF`).

The class also provides methods for:

- **Dynamic status updates** (`update_panel`, `modify_cp_status`, `reset_cp`) to synchronize the UI with the database.

- **Message handling** (`add_app_message`, `add_request_message`) to display system and user notifications.

- **Interaction with Kafka directives** through the `directives_producer` to remotely control charging points.

- **Database integration** via SQLite to retrieve and persist charging point data.

main

The `main.py` module serves as the **entry point** and **orchestration layer** of the entire Electric Vehicle Charging Central system. It integrates all major subsystems—including configuration, communication, monitoring, and the graphical interface—into a unified operational workflow.

At startup, the script loads the **system configuration** via the `CentralConfig` class and initializes essential components such as Kafka producers and consumers (`DirectivesProducer`, `SupplyReqConsumer`, `SupplyInfoConsumer`, etc.), a **monitoring TCP server** (`MonitorServer`), and the **graphical control panel** (`CentralApp`).

Key functionalities:

- **Thread management:** Several background threads handle asynchronous tasks like monitoring connections, consuming Kafka messages, and updating the graphical interface in real time.

- **Message routing:** A shared queue (`gui_queue`) coordinates communication between backend processes and the GUI, ensuring thread-safe message handling.

- **Charging session control:** The system processes supply requests, status updates, and error reports, sending corresponding responses and directives to charging points.

- **Database synchronization:** SQLite is used to maintain consistent status information for all registered charging points.

Upon termination, the program gracefully stops threads and resets the database state.

monitor_handler

The `monitor_handler` module defines the **communication logic** between the central system and individual charging points (CPs) over a structured TCP protocol. It interprets messages received from each CP, manages registration and authorization, and synchronizes real-time status updates with the graphical user interface via a shared message queue.

The core function, `monitor_handler`, continuously listens for and processes incoming JSON-formatted messages using an `STXETXConnection` object. It handles three main message types:

- **`auth`** – verifies whether a charging point is registered in the central database and returns authorization data.
- **`register`** – registers a new charging point in the SQLite database with a randomly assigned price, updating the central interface.
- **`status`** – receives operational status codes (e.g., active, supplying, stopped) and relays them to the GUI for live monitoring.

The module also defines helper functions:

- **`register()`** – inserts a new charging point entry into the database and notifies the GUI.
- **`authorize()`** – checks if a charging point exists and retrieves its current configuration.

Comprehensive exception handling (`ConnectionClosedException`, `ClosingConnectionException`) ensures stable communication and updates disconnection states properly.

monitor_server

The `monitor_server` module implements the **TCP server component** responsible for managing incoming network connections from distributed charging points. It establishes and maintains communication channels that allow real-time data exchange between each charging point and the central system.

The main class, **`MonitorServer`**, encapsulates server initialization and connection handling logic:

- **`__init__`** – creates and binds a TCP socket to a specified IP address and port.

- **`listen()`** – places the server in a listening state, waiting for up to `MAX_CONNECTIONS` concurrent clients.

- **`accept()`** – accepts new connections and spawns a dedicated thread to handle communication with each charging point using the provided `client_handler` function (typically `monitor_handler`).

Each connection is wrapped inside an **`STXETXConnection`** object, which standardizes message framing and ensures reliable data transmission. The use of **multithreading** enables concurrent interaction with multiple charging points without blocking other operations.

run.sh

The **`run.sh`** script automates the deployment and execution of the *Central Application* inside a Docker container. It serves as a lightweight orchestration tool to simplify testing and deployment in different environments, ensuring consistent runtime behavior across systems.

stx_etx_connection

The `STXETXConnection` module implements a structured communication protocol over TCP sockets, enabling reliable message exchange between the central server and distributed charging points. It uses ASCII control characters—such as STX, ETX, ENQ, EOT, ACK, and NACK—to define message boundaries, manage handshakes, and ensure data integrity.

The class encapsulates low-level socket operations, providing methods for:

- Message transmission (`send_message`) with integrity checking using a simple LRC checksum.

- Message reception (`recv_message`), validating messages and acknowledging or rejecting them automatically.

- Connection management (`enq_message`, `enq_answer`, `close`, `eot_message`) for initializing and terminating communication safely.

Custom exceptions (`ConnectionClosedException`, `ClosingConnectionException`) are used to handle network errors and disconnections gracefully.

supply_error_producer

The `SupplyErrorProducer` module implements a **Kafka producer** responsible for reporting supply-related errors in the electric vehicle charging system. It provides a

dedicated communication channel (`supply-error` topic) to notify other system components of abnormal events during a charging session.

The class establishes a connection to a Kafka broker using the specified IP and port. The main method, **`send_error()`**, formats error information—including the affected supply session ID—into a JSON message and publishes it to the Kafka topic. The producer ensures message delivery by flushing the buffer immediately after sending.

supply_info_consumer

The `SupplyInfoConsumer` module implements a **Kafka consumer** responsible for retrieving real-time supply information from the distributed charging points. It subscribes to the `supply-data` topic, ensuring that the central system receives updates about ongoing charging sessions.

The class establishes a Kafka consumer connection with **manual offset management**, allowing precise control over which messages have been processed. The **`get_info()`** method polls messages, decodes them from JSON format, and commits offsets only after successful processing. Errors are handled appropriately, including end-of-partition events and consumer exceptions.

supply_info_producer

The `SupplyInfoProducer` module implements a **Kafka producer** that disseminates real-time charging session information from the central system to other components. It publishes messages to the `supply-data2` topic, providing updates on ongoing consumption and billing.

The class offers three main methods:

- **`repeat_msg()`** – republishes an existing message, supporting message reliability and redundancy.
- **`send_supplying_msg()`** – sends periodic updates during a charging session, including consumption and price.
- **`send_ticket()`** – sends a final summary message at the end of a session, reporting total energy consumed and total cost.

supply_req_producer

The `SupplyReqConsumer` module implements a **Kafka consumer** that retrieves charging session requests from drivers. It subscribes to the `supply-req` topic, allowing the central system to process incoming requests asynchronously and in real time.

The class establishes a Kafka consumer with **manual offset management**, ensuring precise control over message acknowledgment. The **`get_request()`** method polls for messages, decodes them from JSON, and commits the offset only after successful retrieval. It handles errors robustly, including end-of-partition and consumer exceptions.

supply_res_producer

The `SupplyResProducer` module implements a **Kafka producer** responsible for sending responses to driver supply requests. It publishes messages to the `supply-res` topic, indicating whether a request has been authorized or denied, along with the associated reason and supply session ID.

The **`send_response()`** method formats response data as a JSON message and immediately flushes it to the Kafka broker, ensuring timely delivery. This module provides the **asynchronous feedback layer**, allowing the central system to notify drivers about the status of their charging requests in a reliable and scalable manner.

test

This script provides a **simple standalone test harness** for the `SupplyReqConsumer` module. It connects to a Kafka broker using the specified IP and port, subscribes to the `supply-req` topic, and continuously polls for driver supply requests.

EV_CP_E

/test

- test_connection_monitor.py

This script serves as a **basic test harness** for the `MonitorConnection` module. It establishes a connection to a monitoring server running on `localhost` at port `3440` and executes the main communication routine via the `run()` method.

compose.yaml

This snippet defines a **Docker Compose service** for the `cp-engine` container

cp_id

The `CPId` module implements a **simple identifier manager** for charging points. It encapsulates the storage and retrieval of a single charging point ID, providing controlled access through the following methods:

- **`set_id(id: str)`** – assigns a unique identifier to the charging point.
- **`get_id()`** – retrieves the currently assigned identifier.
- **`has_id()`** – checks whether an identifier has been set.

directives_consumer

The `DirectivesConsumer` module implements a **Kafka consumer** designed to receive operational directives from the central system. It subscribes to the `central-directives` topic, allowing individual charging points or all charging points (`target="all"`) to receive control messages.

Key functionalities include:

- **`get_directive()`** – polls messages from Kafka, decodes them from JSON, and filters directives based on the `cp_id` target. Only messages relevant to the specific charging point or broadcasted to all are returned.
- **Automatic offset management** – commits messages upon retrieval to avoid reprocessing, ensuring message consistency.
- **Error handling** – detects end-of-partition and consumer errors to maintain robust operation.

engine_app

The `engine_app` module provides a **command-line simulation interface** for a charging point engine, representing the interaction between an electric vehicle driver and the charging point. Its primary objectives are to manage the **charging process, consumption tracking, and cost calculation**.

Key components:

1. **Connection Management**
    - `try_connexion()` attempts to establish a connection with the central server, handling connection failures gracefully.
2. **Charging Session Simulation**
    - `engine_app(price)` simulates a driver connecting to a charging point.
    - It tracks **elapsed charging time**, calculates **energy consumption** based on a fixed 22 kW semi-fast AC point, and computes the corresponding **cost**.

- The session continues until the driver manually “unplugs” the vehicle (input detection).

3. **Structured Supply Interface**

   - `supply_interface(cp_status: CPStatus, supply_info: SupplyInfo)` integrates with the `CPStatus` and `SupplyInfo` classes to provide a more modular simulation.

   - The charging point updates its operational state dynamically (`supplying → active`) and sends consumption data for central monitoring.

engine_config

The `EngineConfig` module provides a **configuration interface** for the charging point engine, enabling flexible deployment and runtime parameterization via command-line arguments. It is responsible for **centralizing connection parameters, operational metadata, and pricing information**.

Key features:

1. **Command-Line Argument Parsing**

   - Uses Python’s `argparse` library to retrieve required parameters such as Kafka server IP and port, monitor server IP and port, and optional charging point metadata (location and price).

2. **Configuration Storage**

   - Stores parsed values as instance attributes (`kafka_ip`, `kafka_port`, `server_ip`, `server_port`, `location`, `price`) for easy access by other engine components.

3. **Static Helper Method**

   - `_get_engine_config()` encapsulates argument parsing and returns a dictionary of configuration values, promoting modularity and reusability.

engine_data

The `EngineData` module defines a **data container** for a charging point engine, encapsulating essential operational and identification attributes. Its primary purpose is to **maintain the internal state and metadata** of a charging point.

engine

The `Engine` module defines the **core operational component** of a charging point, integrating **status management** with **communication capabilities** to a central monitoring system.

main

Purpose: Orchestrates the operation of an electric vehicle charging point engine, managing state, communication, and energy supply sessions.

Configuration:

- Loads engine parameters: Kafka server, monitor server IP/port, CP location, and energy price.
- Encapsulated in a configuration object for consistent access.

State Management:

- Tracks CP ID, operational status (active, supplying, stopped, waiting), and supply information.
- Status changes are handled through a dedicated status object.

Monitor Communication:

- Listens for incoming monitor connections.
- Receives CP ID and confirms connectivity.
- Ensures central system can control and identify the CP.

Directive Handling:

- Runs a separate thread to listen for directives from the central system.
- Supports actions: start supply, stop, emergency stop, resume.
- Updates CP state and triggers supply sessions or ticket generation.

Manual Supply Interface:

- Allows operator-initiated supply requests without driver app.
- Requests authorization from central system and waits for supply directive.
- Tracks energy delivery: consumption, cost, and end-of-session ticket.

Concurrency & Synchronization:

- Uses threads, locks, and events to safely manage simultaneous directives and operator actions.

monitor_handler

The `monitor_handler` manages communication between a charging point (CP) engine and its monitor. It receives requests and control messages, updates the CP's internal state, and ensures thread-safe operations. The handler maintains an active loop to respond to queries and manage disconnections gracefully.

Key Functions:

- **Initialization:**
  - Receives the CP ID and updates the `EngineData` instance safely.
  - Sets the CP status to active if it was stopped.
  - Sends acknowledgment to the monitor and signals ID reception.

- **Monitoring Loop:**
  - Responds to messages:
    - Health requests → returns CP status.
    - Location requests → returns CP location.
    - Central notifications → updates status (stopped/active).
    - Price updates → adjusts CP's supply price.

- **Thread Safety:**
  - All updates to shared data are protected by a lock.

- **Connection Handling:**
  - Handles graceful closure or unexpected disconnections by stopping the CP and closing the connection.

monitor_server

The `MonitorServer` provides the communication interface between a charging point (CP) engine and its monitor. It manages TCP connections, delegates requests to a handler, and ensures thread-safe operations for the CP's shared data.

Key Components and Functions:

- **Initialization:**
  - Creates a TCP server socket bound to the specified IP address and port.
  - Prepares the server to accept connections (single connection by default).

- **Listening (`listen`):**

    - Puts the server in listening mode, waiting for the monitor to connect.
    - Outputs a status message indicating readiness.

- **Accepting Connections (`accept`):**
    - Accepts an incoming monitor connection.
    - Wraps the socket in an `STXETXConnection` object for protocol handling.
    - Spawns a separate thread to run the provided `client_handler`, passing:
        - The connection object.
        - Shared `EngineData`.
        - A `Lock` for thread-safe operations.
        - Optional `Event` to signal CP ID reception.
- **Closing (`close`):**
    - Safely closes the server socket.

run.sh

This Bash script automates building and running the Docker container for the Charging Point Engine, while allowing configurable network and environment parameters.

supply_data

`SupplyData` provides a simple, encapsulated mechanism to track real-time supply metrics. Each call to `update_supply()` simulates the energy delivered over a fixed interval, updating both consumption and cost accordingly, making it suitable for integration with the engine's reporting and billing logic.

supply_info

The `SupplyInfoProducer` class acts as a Kafka producer responsible for sending real-time energy supply data and final supply tickets to the central system. It enables the engine to communicate energy consumption metrics and billing information reliably.

Key Components and Behavior:

- **Attributes:**
    - `producer`: Instance of `confluent_kafka.Producer` configured with the Kafka server.
      `supply_id`: Unique identifier for the current supply session.
    - `TOPIC_NAME`: Kafka topic name (`supply-data`) where messages are published.

- **Public Methods:**

- `send_supplying_msg(consumption, price)`: Publishes an ongoing supply message including the current consumption and price.
- `send_ticket(total_consumption, total_price)`: Publishes a final supply ticket summarizing the total consumption and cost of the supply session.

supply_req_producer

The `SupplyInfo` class acts as a higher-level abstraction that combines local supply tracking with message publishing. It manages the energy consumption and cost data for a charging session while ensuring that updates are sent to the central system through Kafka.

Key Components and Behavior:

- **Attributes:**
  - `producer`: Instance of `SupplyInfoProducer`, responsible for sending messages to Kafka.
  - `data`: Instance of `SupplyData`, maintaining the current consumption and cost calculations.

- **Public Methods:**
  - `send_info()`: Sends the current supply data (consumption and cost) to the central system and then updates the local consumption and cost counters.
  - `send_ticket()`: Sends the final supply ticket summarizing the total consumption and cost at the end of the sessi

supply_res_consumer

`SupplyReqProducer` abstracts the communication of supply requests. Whenever a CP wants to initiate a charging session, this class generates and sends a structured Kafka message to the central system. It decouples the CP's internal logic from the messaging layer, ensuring that the supply request is reliably transmitted.

EV_CP_M

/test

- test_connection_server

This script demonstrates the instantiation of an `EngineConnection` object, which manages the communication between a client (typically a charging point interface) and the engine module. It is intended as a simple test or entry point to establish a connection.

central_connection

The `CentralConnection` class provides a communication interface between a Charging Point (CP) engine and the Central system. It extends `STXETXConnection`, leveraging a

standardized message protocol to exchange structured JSON messages for authorization, registration, and status updates.

Core Responsibilities:

1. **Connection Management:**
    - Establishes a TCP connection to the Central server on a specified IP and port.
    - Provides methods to initiate (`start_connection`) and gracefully close (`close_connection`) the communication.

2. **Authorization and Registration:**
    - `authorize(cp_id)` sends a JSON message requesting authorization for a specific CP. The Central responds with an authorization status and additional metadata (e.g., pricing).
    - `register(cp_id, location)` sends a registration request for a new CP, receiving confirmation and relevant data from the Central.

3. **Status Reporting:**
    - `send_status_message(current_status)` transmits the CP's current status to the Central system using a structured JSON payload. This ensures the Central is aware of the CP's operational state in real-time.

compose.yaml

This snippet defines a **Docker Compose service** for the `cp-monitor` container

engine_connection

Connection Management:

- Establishes a TCP connection to a monitor server at a specified IP and port.
- Provides methods to start (`start_connection`) and close (`close_connection`) the connection using a handshake protocol.

Health and Status Reporting:

- `req_health_status(status)` requests the current health status from the monitor and updates the CP's local `CPStatus` object accordingly. Status codes are mapped to states such as active, supplying, stopped, waiting for supply, or broken.
- This ensures the CP's internal state remains synchronized with the monitor's view.

Location Retrieval:

- `req_location()` queries the monitor for the CP's physical or logical location. This information can be used for reporting, registration, or display purposes.

Central State Notifications:

- `send_central_fallen()` and `send_central_restored()` inform the monitor of critical events regarding the Central system's availability, allowing the CP to transition safely between operational states.

Price Updates:

- `send_price(price)` sends the current electricity price to the monitor, enabling real-time pricing for energy supply calculations.

main

This module manages a Charging Point (CP) monitor that coordinates communications between the Engine and the Central system, ensuring authorization, registration, and continuous status updates.

Key Functions and Workflow:

1. **Initialization:**
    - Loads configuration via `MonitorConfig` (Central/Engine IPs, ports, CP ID).
    - Creates `CentralConnection`, `EngineConnection`, and `CPStatus` instances.

2. **Connection Setup:**
    - Establishes connections with the Central and Engine.
    - Ensures both are reachable before proceeding.

3. **Authorization & Registration:**
    - Attempts CP authorization with the Central.
    - If failed, retrieves location from Engine and registers CP.
    - Stores electricity price for local Engine synchronization.

4. **Monitoring Loop:**
    - Requests Engine health status and updates `CPStatus`.
    - Sends status to the Central if operational.
    - Handles connection failures: marks CP as broken or continues local operation if Central is down.

monitor_config

The `MonitorConfig` module encapsulates the configuration of a Charging Point (CP) monitor, providing a centralized way to retrieve connection parameters for the Central server, Engine, and the CP itself.

Key Features and Workflow:

1. **Argument Parsing:**
    - Uses Python's `argparse` to read command-line parameters.
    - Mandatory parameters:
        - `--id`: the Charging Point identifier.
        - `--central-ip` and `--central-port`: address of the Central server.
        - `--engine-ip` and `--engine-port`: address of the Engine service.

2. **Configuration Storage:**
    - Parsed arguments are stored as instance attributes:
      `cp_ip`: the CP's unique ID.
        - `central_ip` / `central_port`: connection info for Central.
        - `engine_ip` / `engine_port`: connection info for Engine.

3. **Static Method `_get_monitor_config()`:**
    - Encapsulates the parsing logic.
    - Returns a dictionary of all required configuration parameters.
    - Ensures required parameters are provided, raising errors if missing.

monitor

The `Monitor` class represents a monitoring entity for a Charging Point (CP), responsible for maintaining communication with both the Engine service and the Central server.

run.sh

This Bash script is used to build and launch the Docker container for the Charging Point Monitor. It provides a convenient way to configure runtime parameters for network communication with the Central server and Engine service.

EV_Driver

compose.yaml

This snippet defines a **Docker Compose service** for the `cp-monitor` container

cp_list.txt

List of CP's that are available for a driver

driver_config

This module handles command-line configuration for the driver application, encapsulating essential connection and identification parameters.

Key Features:

1. **Argument Parsing:**
    - Uses `argparse` to capture command-line inputs for:
        - `--id`: Unique identifier for the driver client.
        - `--kafka-ip`: IP address of the Kafka server.
        - `--kafka-port`: Port of the Kafka server.

2. **Encapsulation:**
    - Stores the parsed arguments as instance attributes (`client_id`, `kafka_ip`, `kafka_port`) for convenient access by the driver application.

3. **Static Helper:**
    - `_get_driver_config()` abstracts the parsing logic, returning a dictionary of arguments.

main

This module allows an EV driver to interact with charging points (CPs) using Kafka messaging, handling supply requests, consumption tracking, and session recovery.

Key Functions:

1. **Initialization:**
    - Loads driver settings (`client_id`, Kafka IP/port) via `DriverConfig`.
    - Reads available CPs from `cp_list.txt`.
    - Checks for incomplete supply sessions and recovers state if needed.

2. **Supply Requests:**
    - Sends supply requests to a CP via `SupplyReqProducer`.
    - Receives authorization or denial via `SupplyResConsumer`.

3. **Monitoring Supply:**
    - Tracks live consumption and cost via `SupplyInfoConsumer`.
    - Detects end of supply via ticket messages and prints final consumption/cost.

4. **Recovery:**
    - Uses local files to resume interrupted supply sessions.
    - Ensures accurate tracking and billing.

5. **User Loop:**
    - Lists CPs, allows manual or automatic selection, executes supply, and repeats until `quit`.

recover_last_data

This module retrieves the most recent Kafka message related to a specific supply session for a driver, enabling recovery of ongoing or recently completed EV charging sessions.

Workflow:

1. **Consumer Setup:**
    - Creates a temporary Kafka consumer using the driver's configuration.
    - Subscribes to the `_SUBSCRIBED_TOPIC` (`supply-data2`).

2. **Polling Loop:**
    - Continuously polls Kafka messages with a 1-second timeout.
    - Stops after 3 consecutive empty polls to avoid indefinite blocking.

3. **Message Processing:**
    - Skips messages with errors or partition EOFs.
    - Decodes JSON payloads and filters messages by the `supply_id`.
    - Tracks the most recent message by comparing Kafka timestamps.

4. **Return Value:**
    - Returns the latest message for the requested `supply_id`, or `None` if no message is found.

run.sh

Automates building and running the driver Docker container with persistent recovery storage.

supply_error_consumer

Consumes Kafka messages about supply errors related to a specific charging session (`supply_id`) for a driver.

Key Features:

1. **Initialization:**
    - Connects to a Kafka broker using `kafka_ip` and `kafka_port`.
    - Sets the consumer group to the driver's ID.
    - Subscribes to the `supply-error` topic.

2. **Methods:**
    - `get_error()`:
        - Polls Kafka for a new message.
        - Ignores messages not matching the `supply_id`.
        - Returns the error message as a dictionary if relevant, otherwise `None`.
        - Handles partition EOF and Kafka errors gracefully.

- `close()`: Closes the consumer safely.

supply_info_consumer

Consumes Kafka messages containing live supply data (energy consumption and cost) for a specific charging session (`supply_id`) tied to a driver.

Key Features:

1. **Initialization:**
   - Connects to a Kafka broker using `kafka_ip` and `kafka_port`.
   - Assigns the driver's ID as the consumer group.
   - Subscribes to the `supply-data2` topic.

2. **Methods:**

   - `get_info()`
     - Polls Kafka for new messages.
     - Ignores messages not related to the current `supply_id`.
     - Returns the supply data as a dictionary or `None` if no relevant message is found.
     - Handles partition EOF and Kafka errors gracefully.
   - `close()`

supply_req_producer

Sends supply requests from a driver to the central system via Kafka, asking permission to start a charging session at a specific charging point (`cp_id`).

Key Features:

1. **Initialization:**
   - Connects to a Kafka broker using `kafka_ip` and `kafka_port`.
   - Sets up a producer for the `supply-req` topic.

2. **Methods:**

   - `send_request(cp_id, driver_id)`
     - Builds a JSON message containing the target charging point ID and the requesting driver ID.
     - Sends the message to the `supply-req` Kafka topic.
     - Flushes the producer to ensure delivery.

supply_res_consumer

Receives responses from the central system regarding supply requests previously sent by a driver. Essentially, it listens for whether the request to start charging has been approved or denied.

Key Features:

1. **Initialization:**
    - Connects to a Kafka broker at `kafka_ip:kafka_port`.
    - Sets up a consumer for the `supply-res` topic.
    - Uses the `driver_id` to filter responses intended for this specific driver.

2. **Methods:**

    - `get_response()`
        - Polls Kafka for messages.
        - Filters messages to only return those with `applicant_id` matching the driver.
        - Handles Kafka errors and returns `None` if no relevant message is found.

    - `close()`

test

Demonstrates how to send a supply request to a charging point via Kafka using the `SupplyReqProducer` class.

Deployment

Kafka

EV_Central

EV_CP_E

EV_CP_M

EV_Driver

Interface

EV_Central

EV_CP_E

EV_CP_M

EV_Driver