Documentation Ev-Charging

Course: Distributed System

Student: Emily Apitzsch

Student: Johanna Helena Baarssen

Date: 14.12.2025

Universidad de Alicante

1. Project Description:

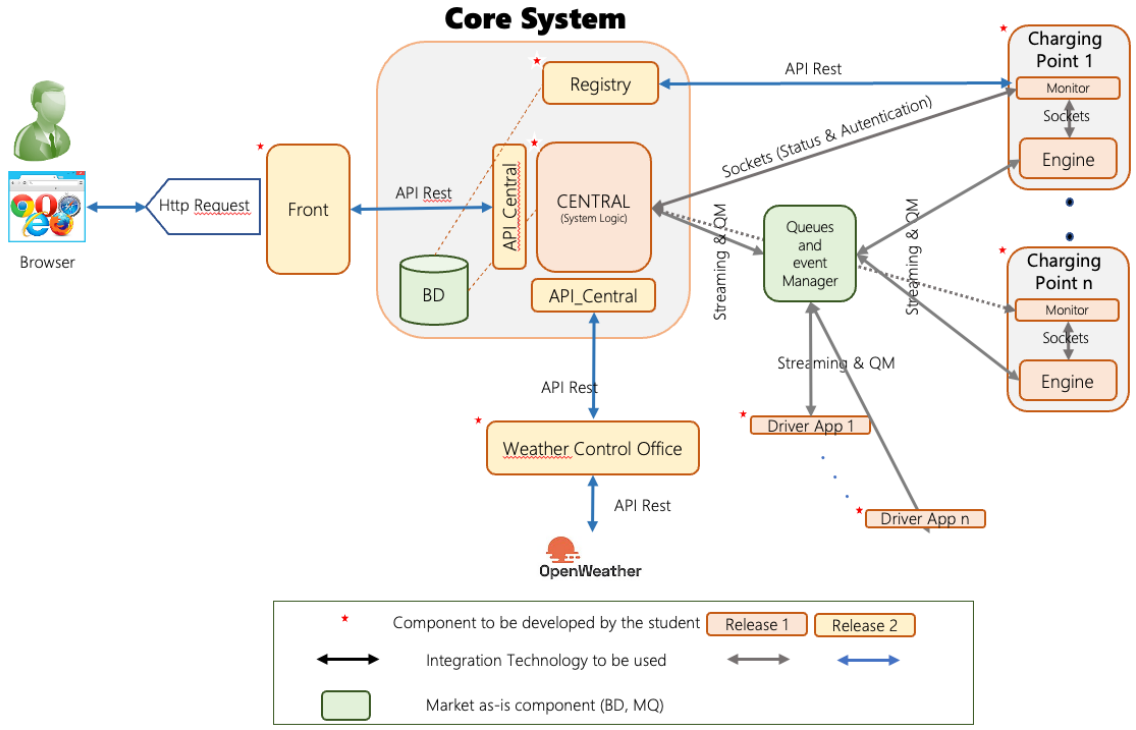
This project implements a distributed Electric Vehicle (EV) Charging System designed to simulate the operation, monitoring, and control of multiple charging points. The main goal of the system is to demonstrate key concepts of distributed system.

The system allows drivers to start and stop charging sessions, administrators to monitor and control charging points, and external services (such as weather providers) to influence system behavior. All components interact through well-defined interfaces, ensuring scalability, resilience, and traceability. The project was developed as part of the Distributed Systems course and focuses on correctness, robustness, and architectural clarity.

2. System Architecture Overview:

The EV Charging System follows a centralized coordination model supported by distributed components.
At the core of the architecture is the Central Server, which acts as the decision-making authority and communication hub.

Surrounding the Central Server are independent components that interact using different communication technologies, including TCP sockets, REST APIs, Kafka message queues, and HTTPS. This hybrid approach reflects real-world distributed systems, where different protocols are chosen depending on latency, reliability, and security requirements.



3. Main System Components:

3.1 EV_Registry

The EV_Registry is responsible for secure registration and authentication of Charging Points and Monitors.

Before interacting with the Central system, each Charging Point and Monitor must:

- register with the Registry using HTTPS,
- provide valid credentials,
- and be marked as active in the registry database.

This design prevents unauthorized devices from joining the system and ensures that all operational components are known and verified before use.

3.2 Central Server

The Central Server is the core component of the system and coordinates all other services. Its responsibilities include:

- managing charging sessions,
- tracking the state of each Charging Point,
- handling driver requests,
- processing telemetry and monitoring data,
- reacting to weather alerts,
- and maintaining a centralized audit log.

The Central Server communicates with different components using:

- TCP for real-time interaction with drivers and charging points,
- Kafka for asynchronous telemetry, status updates, and commands,
- REST APIs for frontend interaction and external services.

3.3 Charging Point (Engine & Monitor)

Each Charging Point is implemented using two independent processes**:**

- **Engine**
  The Engine simulates the charging process. It is responsible for:
  - executing charging sessions,
  - sending telemetry data (energy, cost),
- **Monitor**
  The Monitor supervises the Engine's availability. It periodically checks whether the Engine is alive and reports its status to the Central system.

This separation increases fault tolerance: even if the Engine fails, the Monitor can still report the failure.

3.4 Driver

The Driver component represents an EV user interacting with the system. Drivers connect to the Central Server using TCP and can:

- authenticate themselves,
- request the start of a charging session,
- stop an ongoing session,
- receive a final charging ticket containing consumed energy and cost.

Drivers do not communicate directly with Charging Points. All interactions are mediated by the Central Server.

3.5 Weather Service (EV_W)

The Weather Service simulates an external dependency that influences system behavior. It periodically retrieves real-time weather data from a public weather API and sends:

- weather updates,
- and weather alerts (e.g., storms or extreme conditions)

3.6 Frontend Dashboard

The Frontend Dashboard provides a visual overview of the system state. It allows administrators to:

- view all Charging Points and their current status,
- show ongoing and historical charging sessions,
- observe weather alerts,
- and issue administrative commands such as out and resume.

The dashboard communicates with the Central Server through a REST API and serves as the primary administrative interface.

3.7 Audit Logging

All security-relevant and operational events are recorded in a centralized audit log stored in the Central database. Logged events include:

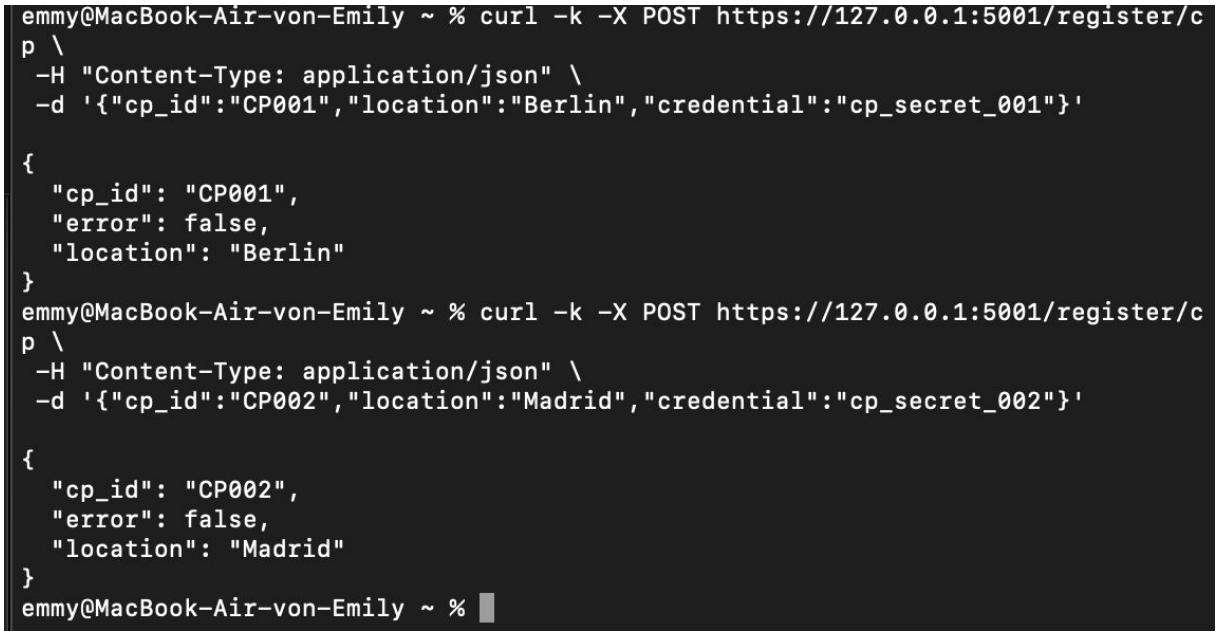
- authentication attempts,
- session lifecycle events,
- administrative commands,
- weather alerts,

- and system failures.

The audit log enables traceability, debugging, and accountability, which are essential properties of distributed systems.

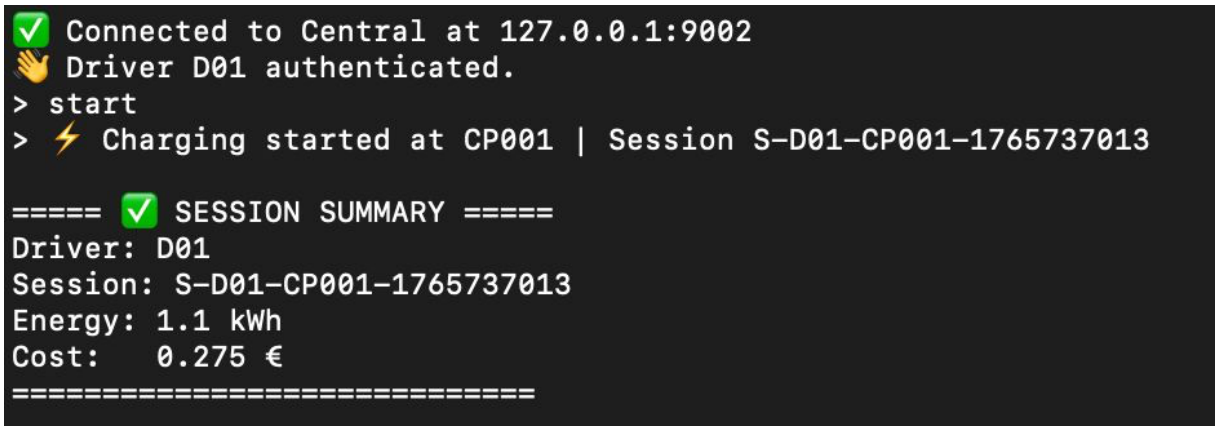
3.8. Screenshots:

Registering Charging Points at the Registry



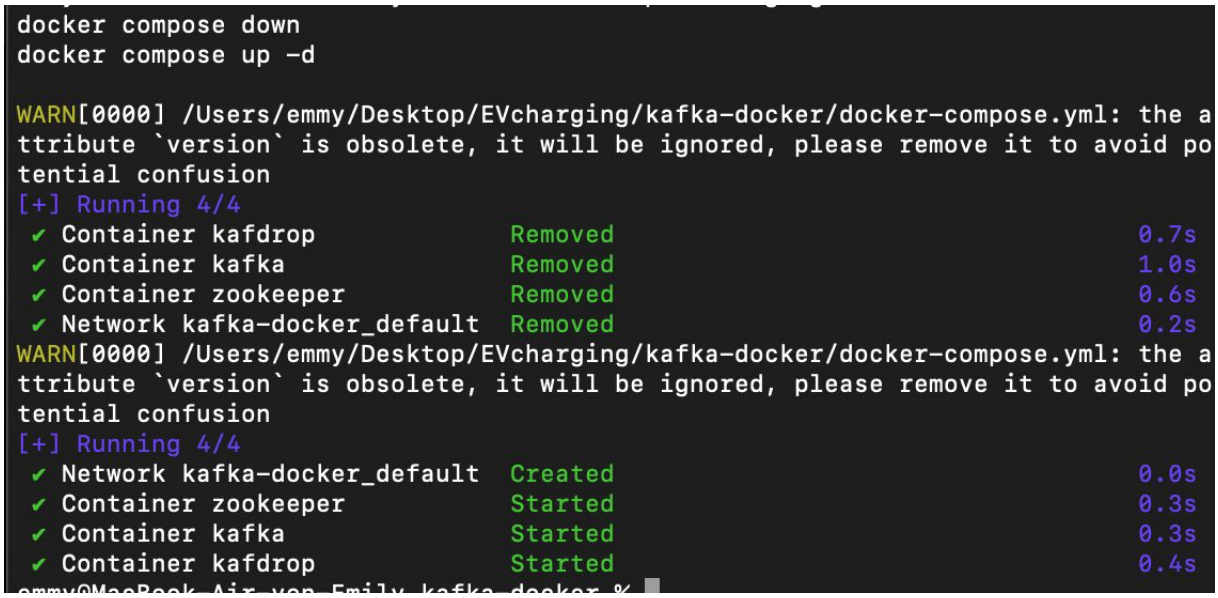
The image above shows the manual registration of Charging Points (CPs) in the EV Registry using HTTPS requests.
Each CP is registered with a unique cp_id, location, and credential.
This step is required so that the Central system can later authenticate Charging Points securely.
Without successful registration, a CP is not allowed to connect or start charging sessions.

Driver: Charging Session with Ticket



This image shows a Driver client successfully starting a charging session and receiving a session ticket.
The ticket confirms that the driver has been authenticated by the Central system and assigned to an available Charging Point.
This demonstrates correct interaction between the Driver and Central components.

Starting Kafka



This image shows Kafka (including Zookeeper and Kafdrop) being started using Docker Compose.
Kafka acts as the messaging backbone of the system and is responsible for transporting telemetry data and alerts.
All other system components depend on Kafka being available before they can start correctly.
Starting the Registry

```
python -m EV_Registry.registry_app

EV_Registry running on https://localhost:5001
 * Serving Flask app 'registry_app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
 Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on https://127.0.0.1:5001
 * Running on https://192.168.1.144:5001
Press CTRL+C to quit
 * Restarting with stat
EV_Registry running on https://localhost:5001
 * Debugger is active!
 * Debugger PIN: 124-085-684
```

This image shows the EV Registry service running successfully.
The registry provides secure HTTPS endpoints for registering Charging Points and Monitors. It is a mandatory component for authentication and ensures that only known components can participate in the system.

Starting EV_W (Weather Service)

```
python -m weather.EV_W

[EV_W] Weather Service started (OpenWeather)
[EV_W] Madrid: 10.12°C, wind 2.06
[EV_W] Berlin: 5.62°C, wind 5.81
[EV_W] Madrid: 10.12°C, wind 2.06
[EV_W] Berlin: 5.62°C, wind 5.81
```

This image shows the EV_W weather service being started.
The weather service periodically provides weather data that can be used by the Central system and frontend dashboard.
This demonstrates integration of an external data source into the system.

Central API: Weather Endpoint

```
127.0.0.1 - - [14/Dec/2025 20:12:44] "GET /api/locations HTTP/1.1" 200 -
127.0.0.1 - - [14/Dec/2025 20:12:44] "POST /api/weather_update HTTP/1.1" 200 -
127.0.0.1 - - [14/Dec/2025 20:12:44] "POST /api/weather_update HTTP/1.1" 200 -
127.0.0.1 - - [14/Dec/2025 20:12:54] "GET /api/locations HTTP/1.1" 200 -
127.0.0.1 - - [14/Dec/2025 20:12:55] "POST /api/weather_update HTTP/1.1" 200 -
127.0.0.1 - - [14/Dec/2025 20:12:55] "POST /api/weather_update HTTP/1.1" 200 -
127.0.0.1 - - [14/Dec/2025 20:13:05] "GET /api/locations HTTP/1.1" 200 -
127.0.0.1 - - [14/Dec/2025 20:13:05] "POST /api/weather_update HTTP/1.1" 200 -
127.0.0.1 - - [14/Dec/2025 20:13:05] "POST /api/weather_update HTTP/1.1" 200 -
127.0.0.1 - - [14/Dec/2025 20:13:15] "GET /api/locations HTTP/1.1" 200 -
```

This image shows the Central REST API responding with weather data.
The API exposes aggregated system data (such as weather and CP status) to the frontend.
This confirms that the Central backend and weather service are correctly connected.

Dashboard Session History

Dashboard Sessions

Session History

Session	CP	Driver	Start	End	kWh	€
S-D01-CP001-1765738332	CP001	D01	1765738332.327581	1765738336.533442	0.5	0.125
S-D01-CP001-1765738314	CP001	D01	1765738314.208438	1765738324.305393	1.1	0.275
S-D01-CP001-1765737329	CP001	D01	1765737329.147101	1765737333.720803	0.5	0.125
S-D01-CP001-1765737308	CP001	D01	1765737308.768779	1765737318.844636	1.1	0.275

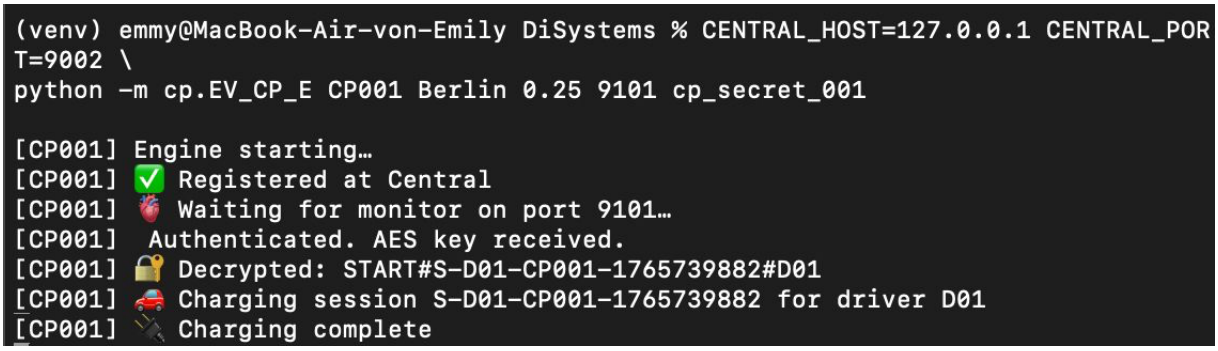
This image displays the frontend dashboard showing historical charging sessions.
Each session contains information such as CP ID, driver ID, energy usage, and cost.
This confirms that session data is stored in the database and correctly visualized.

Starting Monitor and Status Updates

```
[MON-CP001] Authenticated at Registry ✓
[MON-CP001] Authenticated at Central ✓
[MON-CP001] Monitor running, engine = 127.0.0.1:9101
[MON-CP001] CONNECTED to Engine ✓
[MON-CP001] STATUS OK
[MON-CP001] STATUS OK
[MON-CP001] STATUS OK
[MON-CP001] STATUS OK
[MON-CP001] Engine LOST ×: 0 bytes read on a total of 1 expected bytes
[MON-CP001] STATUS DISCONNECTED
```

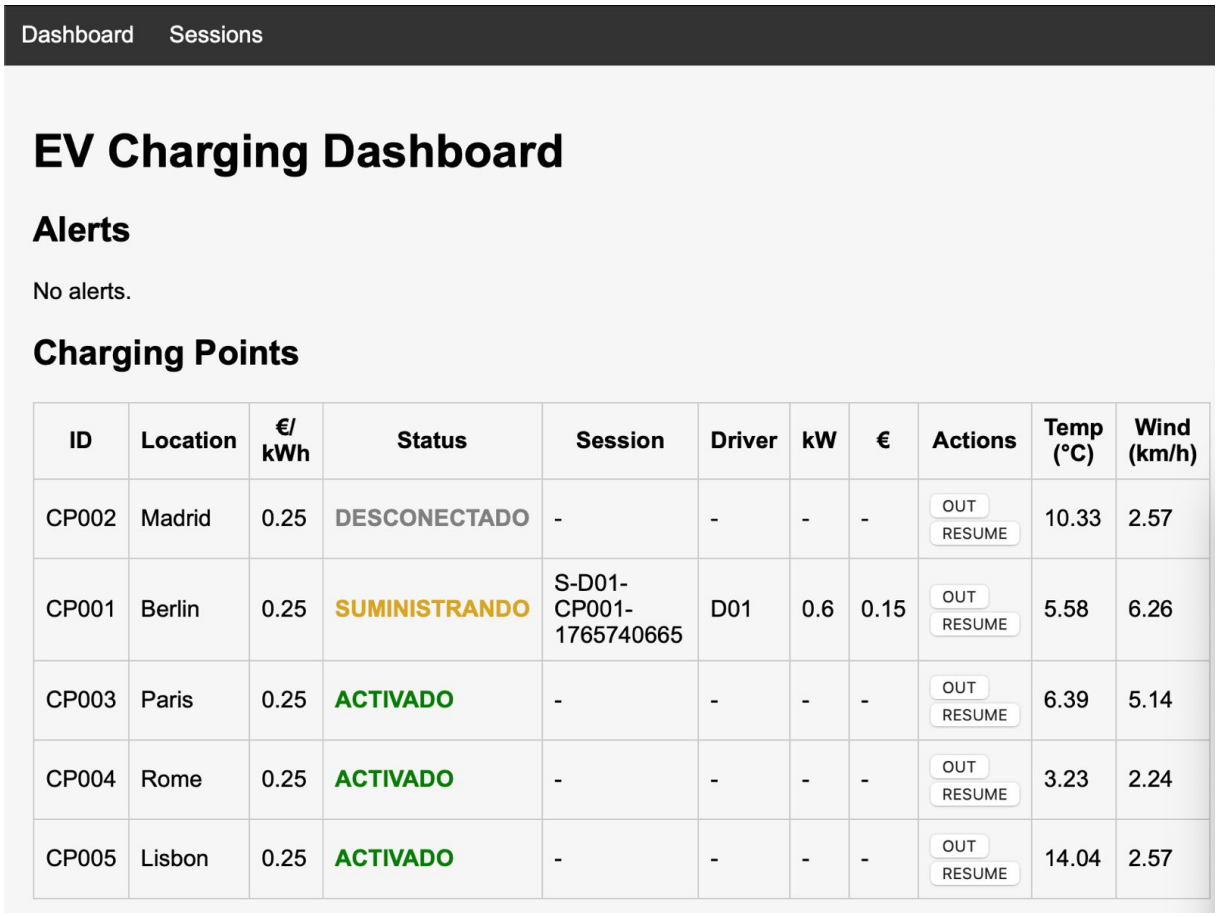
This image shows a Charging Point Monitor connecting to its Engine and sending status updates.
The Monitor checks the health of the Charging Point and reports availability or failures.
This fulfills the monitoring requirement of the system architecture.

Starting Engine and Charging Session Updates



This image shows a Charging Point Engine running and producing charging telemetry.
During an active session, the engine sends power usage and cost updates via Kafka.
This confirms that the simulation of real-time charging works as intended.

Dashboard with One Disconnected and One Charging CP



This image shows the dashboard visualizing different CP states.
One Charging Point is disconnected, while another is actively charging.
This demonstrates real-time system monitoring and correct handling of failures.

Driver When No Charging Point Is Free

```
start
> ⚡ Charging started at CP005 | Session S-D01-CP005-1765741038

===== ✅ SESSION SUMMARY =====
Driver: D01
Session: S-D01-CP005-1765741038
Energy: 1.1 kWh
Cost:   0.341 €
============================== start
>  No charging point available at the moment.
```

This image shows a Driver being denied a charging session because no Charging Point is available.

The Central system correctly enforces resource availability rules.
This confirms correct error handling and business logic.

EV_Weather

```
[EV_W] Madrid: 10.3°C, wind 2.57
[EV_W] Berlin: 5.63°C, wind 6.26
[EV_W] Paris: 6.4°C, wind 5.14
[EV_W] Rome: 2.85°C, wind 2.24
[EV_W] Lisbon: 14.06°C, wind 3.09
[EV_W] Madrid: 10.3°C, wind 2.57
[EV_W] Berlin: 5.63°C, wind 6.26
[EV_W] Paris: 6.4°C, wind 5.14
[EV_W] Rome: 2.85°C, wind 2.24
[EV_W] Lisbon: 14.06°C, wind 3.09
[EV_W] Madrid: 10.3°C, wind 2.57
[EV_W] Berlin: 5.63°C, wind 6.26
[EV_W] Paris: 6.4°C, wind 5.14
[EV_W] Rome: 2.85°C, wind 2.24
[EV_W] Lisbon: 14.06°C, wind 3.09
[EV_W] Madrid: 10.3°C, wind 2.57
[EV_W] Berlin: 5.63°C, wind 6.26
[EV_W] Paris: 6.4°C, wind 5.14
[EV_W] Rome: 2.85°C, wind 2.24
[EV_W] Lisbon: 14.04°C, wind 2.57
```

This image shows the EV_Weather component running and producing weather updates.
The weather data is integrated into the Central system and frontend.
This demonstrates successful distributed communication between services.

4. Security:

Security is important in our EV Charging System because several independent components communicate over the network.Charging Points and Monitors first have to register at the EV_Registry. This registration is done via HTTPS, which means that asymmetric encryption (public and private keys) is used automatically to protect the communication. This ensures that credentials cannot be read by third parties during registration.

After a Charging Point has been successfully registered, it connects to the Central server and receives a symmetric key. From that point on, communication between the Central system and the Charging Point is encrypted using symmetric encryption.

Drivers never communicate directly with Charging Points. All requests go through the Central system, which makes access control easier and reduces potential security risks.

By combining asymmetric encryption between the Registry and Charging Points with symmetric encryption between the Central system and Charging Points, the system achieves a good balance between security and simplicity without adding unnecessary complexity.

5. Deployment Guide:

The following commands were used to start the system up.

Common setup:

cd ~/Desktop/EVCharging/DiSystems

python3 -m venv venv

source venv/bin/activate

pip install -r common/requirements.txt

pip install pycryptodome

Network setup:

ipconfig getifaddr en0

"kafka_bootstrap": "172.20.10.12:9092"

Start kafka:

cd ~/Desktop/EVcharging/kafka-docker

docker compose up -d

Start Registry

python -m EV_Registry.registry_app

Start Central System

python -m central.EV_Central

Start Charging Point Engines

python -m cp.EV_CP_E CP001 Berlin 0.25 9101 cp_secret_001

Start Monitors

python -m cp.EV_CP_M 127.0.0.1 9101 CP001 cp_secret_001
Start Drivers

python -m driver.EV_Driver --central-ip 127.0.0.1 --central-port 9002 --driver-id D01

6. Conclusion:

In this project, we built and deployed a distributed EV Charging Network that runs across multiple physical machines. The different components communicate with each other using TCP, REST, HTTPS, and Kafka, which allowed the system to work reliably even when several parts were running at the same time.

Key features such as Charging Point registration, authentication, driver authorization, real-time telemetry, monitoring, and visualization were implemented and tested step by step. The system is set up in a modular way, making it easier to scale, maintain, and understand the role of each service.

Overall, the project meets the main goals of a distributed system by combining communication, messaging, security, monitoring, and data storage into a working and well-structured solution.