UNIVERSITY OF PISA

MSc in Computer Engineering

Internet of Things

Smart Lithium-ion Battery Energy Storage Management

Professors:

Prof. Giuseppe Anastasi

Prof. Carlo Vallati

Ing. Francesca Righetti

Student:

Andrea Di Matteo

ACADEMIC YEAR 2024/2025

Contents

1 Introduction

Due to the increasing usage of Distributed Energy Resources (DERs) powered by renewable sources, both in domestic PhotoVoltaic (PV) and industrial Eolic and PV systems, the modern electrical grid landscape is experimenting a paradigm shift from centralized generation to decentralized microgrids. These are essential to increase resilience and enabling the integration of intermittent renewable sources, such as solar photovoltaic and wind energy. In this context, Energy Storage Systems (ESS) have emerged as critical components for limiting the consequences of the stochastic nature of these sources, ensuring a reliable power supply even when renewable sources are not present or limited. Recent comparative analyses identify **Lithium-ion Battery Storage Systems (LBSS)** as the most effective solution for ESS, offering the optimal balance between technical performance and economic viability.

However, the efficiency of ESSs is highly dependent on the underlying control logic. There are two main operational challenges in managing LBSSs: The firs one concerns the **manteinance and verification of the battery State of Health (SoH)**. As Lithium-ion cells are subject to non-linear degradation, applying the best practices for maximizing battery lifespan and continuous monitoring is fundamental to track the operating characteristic of the system and identifying the need for maintenance before significant degradation occurs.

The second main challenge concerns the **optimal scheduling of energy flows** between the LBSS, the renewable source and the main grid. The control system must determine in a predictive way the optimal charge and discharge rates for each battery. This mechanism is crucial not only for mitigating peak overloads but also for ensuring the economic viability of the microgrid.

To address this challenges, this work proposes a Smart IoT system that provides a framework for monitoring and controlling LBSSs that leverages Machine Learning (ML) algorithms deployed at the edge for real-time maintenance and energy flows scheduling.

2 Architecture

The problem of optimal ESS management shares common challenges, wether applied to Electricity Suppliers, managing multiple production and storage sites, or single domestic environments; from a high level point of view a house with its PV and LBSS is no different from an electricity production site with its LBSS, the only difference will be the scale of the energy produced.

The core interface in both cases is the **inverter**, in modern systems, hybrid inverters are used to manage the DC-AC flow from solar panels to the power grid or loads and the bi-directional DC-DC flow to charge and discharge the batteries, Figure 2.1 graphically represents the behavior of this device. Physical routing of power is handled by programmable **relays**. These devices are critical for safety operations such as isolating the batteries in case of fault, and Demand Side Management (DSM) strategies, such as load shedding.

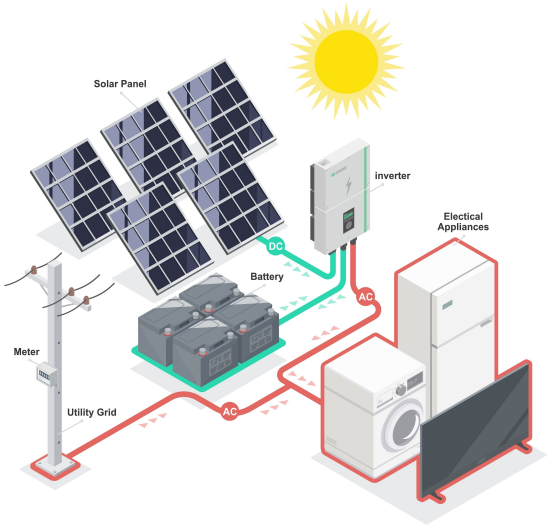

Figure 2.1: Main behaviors of an Hybrid inverter, there are 2 main flows: DC-AC from batteries or renewable sources to power grid or home appliances, DC-DC from/to batteries for charge and discharge

In this context, classical hybrid inverters operate with rigid, pre-programmed logic (e.g. `if battery % > X then do something`), the system proposed in this work instead

is based on an automatic control paradigm designed for safety-critical operations and optimal power flows scheduling for maximizing LBSS lifespan.

2.1 System Architecture

The overall architecture is stratified into three logical levels:

1. **The Edge Layer:** Composed of **BatteryController**, responsible for the real-time safety and health monitoring of a single Lithium-ion Battery Pack.
2. **The Aggregation Layer:** Composed of the **uGridController**, connected to the local energy inverter and a cluster of local BatteryControllers. The uGridController schedules the energy flows for charge and discharge of batteries and acts as a proxy between the BatteryControllers and the Remote Control Applications. This layer also includes the **RPL-border-router** that interfaces the WSN to the Internet.
3. **The Cloud Layer:** The main actors of the Cloud Layer are a **MySQL database** instance that implements data persistence, an **MQTT broker** for asynchronous communications to the Client Application and a **Remote Control Application (RCA)** that reads information about actuators and sensors, implements simple closed loop logic and data persistence storing historical data on the database.

This logical separation ensures that latency-critical operations (e.g. fault isolation) are executed in real-time directly at the batteries, while computationally intensive or latency-tolerant tasks are offloaded to higher layers. The Wireless Sensor Network (WSN) application level stack is based on CoAP for lightweight and efficient communication. Figure 2.2 presents the overall system architecture.

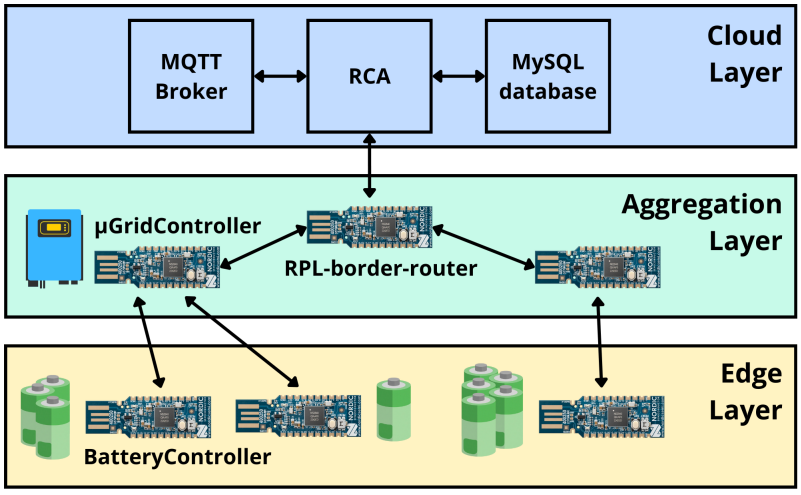

Figure 2.2: Architecture of the overall system: the blue layer represents the Cloud layer, the green layer represents the Aggregation Layer, the yellow layer represents the Edge Layer. The Aggregation Layer and the Edge Layer are WSNs

2.2 BatteryController

The BatteryController is the sensor node of a single Lithium-ion battery pack. Implemented on a nRF52840 dongle, its primary objective is to monitor the current **State of Health (SoH)** of the batteries, perform safety-critical operations such as isolating the batteries from the microgrid in case of faults and actuating the power flow from/to the batteries, the overall architecture of the BatteryController is depicted in Figure 2.3

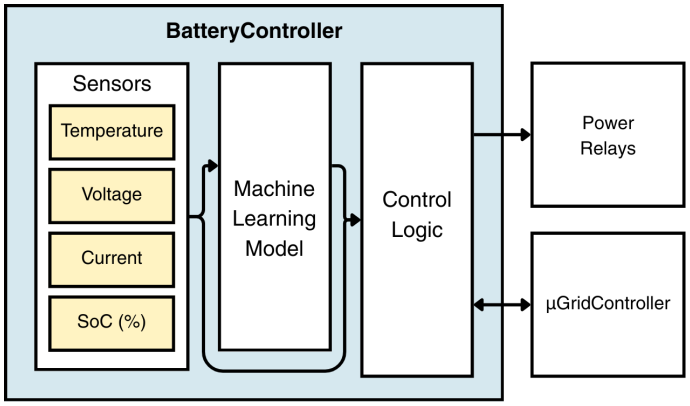

Figure 2.3: Architecture of the BatteryController. This device is capable of receiving commands from the uGridController, send its current state to the latter and actuate the Power Relays based on internal control logic.

2.2.1 State of Health (SoH)

The State of Health (SoH) is a key metric used to quantify the current physical condition of a battery compared to its nominal state at the beginning of its life cycle. It is defined as the ratio of the current **Battery Capacity** ($C_{current}$) to its ideal one ($C_{nominal}$), expressed as a percentage. Historically, battery capacity is measured in *Ampere-hours*, representing the current that the battery is able to deliver over a specific period.

$$SoH(t) = \frac{C_{current}}{C_{nominal}} \times 100\% \tag{2.1}$$

As the battery ages due to oxidation and stress from charge/discharge cycles, its effective capacity diminishes, and its internal resistance increases. Moreover, in order to accurately evaluate the current battery capacity, a **Full Discharge Test** should be performed, i.e., charging the battery to 100% and discharging it to 0% while measuring the time taken and the current delivered.

However, this operation, which is the gold standard for industrial evaluation, is not feasible in real systems such as electric cars and domestic ESS. This motivates the use of ML models to estimate the SoH with acceptable error rates using real-time

sensor data.

To achieve this, the BatteryController runs a local Machine Learning (ML) model that processes a sliding window of the last 20 Voltage, Current, Temperature, and State of Charge (as a percentage) measurements to estimate the battery's SoH in real-time. An high level view of the ML model's dataflow is depicted in Figure 2.4. The model was trained on the NASA Battery Dataset. However, this dataset is based on low-voltage batteries (4.2V); for simplicity, values in this project are normalized to utilize this model, given that public datasets on LBSS charge/discharge cycles are not available.

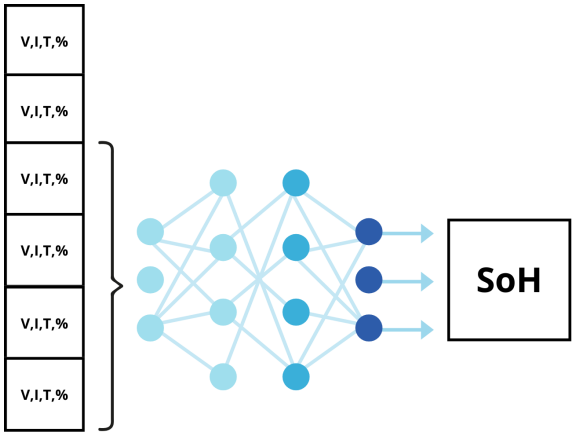

Figure 2.4: Dataflow of the Edge Machine Learning model for SoH estimation.

By inferring the degradation level directly at the edge, the controller is capable of autonomously triggering safety-critical mechanisms with milliseconds response times.

2.2.2 Power Relay

To execute commands and ensure system safety, the BatteryController manages a physical actuator, namely a **Power Relay**. This component acts as the primary control unit for the battery, determining its state based on three key inputs:

- Real-time telemetry (sensor data);
- SoH estimation provided by the local ML model;
- Commands sent from the uGridController.

Specifically, the relay operations can be categorized into three main functions:

1. **Safety Isolation:** Disconnecting the battery from the circuit in critical scenarios. This occurs if the SoH drops below a viable threshold (indicating a need for maintenance), if the temperature exceeds safety limits, or if state values deviate significantly from nominal operating ranges.
2. **Charging:** Closing the circuit to allow DC current flow from the hybrid inverter to the battery pack.

3. **Discharging:** Closing the circuit to allow DC current flow from the battery to the inverter for grid or load supply.

Although the uGridController can send arbitrary commands to toggle the relay for energy management purposes, its authority is limited by the safety logic embedded in the BatteryController's firmware. Local safety constraints always have higher priority over remote commands. Furthermore, when the battery is isolated due to safety such as low SoH, the system enters an isolation state that requires manual intervention for maintenance or battery replacement.

2.3 uGridController

The uGridController is the sensor node attached to a hybrid inverter and connected to a cluster of Lithium-ion battery packs. Also implemented on an nRF52840 dongle, its primary objective is to schedule optimal charge and discharge rates for minimizing Lithium-ion batteries capacity degradation, the overall architecture of the uGridController is depicted in Figure 2.5

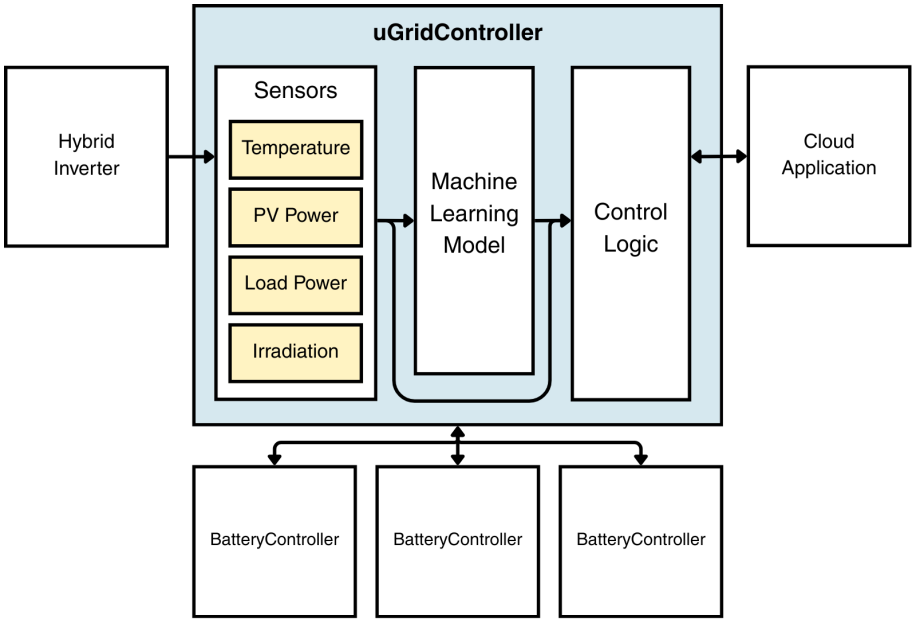

Figure 2.5: Architecture of the uGridController. This device collects data from internal sensors and the Hybrid Inverter, receives the BatteryControllers states and sends updates on power flows to the batteries

2.3.1 Predictive Strategy

To schedule an optimal charge and discharge flow between the batteries, the uGridController adopts a strategy based on predicted power produced by the renewable sources and predicted load power. To achieve this, the uGridController runs a local ML model that processes a sliding window of the last 20 PV power, Load power,

Temperature and Irradiation also considering the hour of the day. At a high level, the dataflow of this ML model is exactly the same of the one in Figure 2.4. Furthermore, given the lack of publicly available datasets that simultaneously capture high-resolution domestic consumption, photovoltaic generation, and local weather conditions, the experimental validation relies on proprietary telemetry extracted from a residential PV system.

2.3.2 Model Predictive Control (MPC)

To translate predicted load and production powers into optimal control commands, the uGridController implements a Model Predictive Control (MPC) strategy.

Periodically, the controller solves an optimization problem to find the optimal battery power flow $u = [u_1, u_2, \ldots, u_n]$ (positive for charging and negative for discharging) that minimizes a multi-objective cost function $J(u)$, subject to physical and safety constraints.

The formulation of $J(u)$ as a weighted sum of different costs is the following

$$J = \alpha \cdot E(u) + \beta \cdot D(u) + \gamma \cdot C(u) \tag{2.2}$$

where α, β and γ are the hyperparameters that determine the relative importance of each objective, these can be dynamically programmed to make one component matter more than another.

- **Grid Usage Term** $E(u)$: this term represents the economical cost of exchanging power with the grid. The power exchanged with the grid is given by the balance equation $P_{grid} = P_{load} - P_{PV} + u$, if we assume a constant power price $Price$ for buying and selling the cost will be:

$$E(u) = Price \cdot (P_{load} - P_{PV} + u) \tag{2.3}$$

  In this context the P_{load} and the P_{PV} terms are the predicted load and PV power from the ML model. The derivative of $E(u)$ with respect to u of $E(u)$ is

$$\frac{\delta E}{\delta u} = Price \tag{2.4}$$

- **Battery Usage Term** $D(u)$: this term accounts for preventing aggressive battery usage, The more frequent the power exchange with the batteries the faster their degradation, for this reason $D(u)$ is defined as:

$$D(u) = u^2 \tag{2.5}$$

The derivative with respect to u is

$$\frac{\delta D}{\delta u} = 2 \cdot u \tag{2.6}$$

- **Lifespan Preservation Term** $C(u)$: This component accounts for the degradation of the battery health. Lithium-ion batteries are prone to accelerated oxidation when operating at the limits of their characteristics (i.e., approaching 0% or 100%). To mitigate this, we define a reference state of charge, $SoC_{ref} = 50\%$, the objective is to minimize the deviation of the **predicted future state of charge** from this reference.

  The dynamic of the battery state of charge is modeled as $SoC_{next} = SoC_{current} + k \cdot u$, where the conversion factor k is defined as $k = \frac{\eta \Delta t}{C_{nom}}$.

  Consequently, the cost term is formulated as the squared error between the predicted and reference states:

$$C(u) = (SoC_{next} - SoC_{ref})^2 = (SoC_{current} + k \cdot u - SoC_{ref})^2 \tag{2.7}$$

  The gradient of this term with respect to the control input u, is derived as:

$$\frac{\partial C}{\partial u} = 2k \cdot (SoC_{current} + k \cdot u - SoC_{ref}) \tag{2.8}$$

Based on the individual components above, the gradient of the overall cost function $J(u)$ is defined as the weighted sum of the different gradients:

$$\nabla J(u) = \alpha \cdot Price + 2\beta \cdot u + 2\gamma k \cdot (SoC_{current} + k \cdot u - SoC_{ref}) \tag{2.9}$$

To solve this optimization problem directly on the resource-constrained nRF52840 dongle, a lightweight **Projected Gradient Descent (PGD)** algorithm is employed. The gradient of the cost function is analytically derived and hard-coded into the firmware, eliminating the need for complex differentiation, the different coefficients α, β, γ and $Price$ can be dynamically adjusted via the endpoint `/ctrl/mpc` of the uGridController. This approach ensures an optimal control policy with minimal memory footprint and execution latencies, making it highly suitable for embedded real-time operation.

2.3.3 Objectives

This feature allows for the manual adjustment of the battery state of charge, which is essential for safety procedures such as fully discharging a battery before disconnection. The system handles this via the `/ctrl/obj`: when an objective is set, the uGridController adds a fixed constraint to the PGD optimization, for instance,

$u = -5$kW for discharging. The front end also computes an ETA based on the latest SoC measurements.

2.4 RPL-border-router

The RPL-border-router is the bridge between the IEEE 802.15.4 Low-Power Wireless Personal Area Network (6LoWPAN) and the external IPv6 network. It is implemented on a dedicated nRF52840 dongle.

2.5 Remote Control Application (RCA)

The Remote Control Application acts as the **Application Gateway** and data acquisition backend. For educational purposes it is a Python Script running on the host machine, its high level architecture is depicted in Figure 2.6.

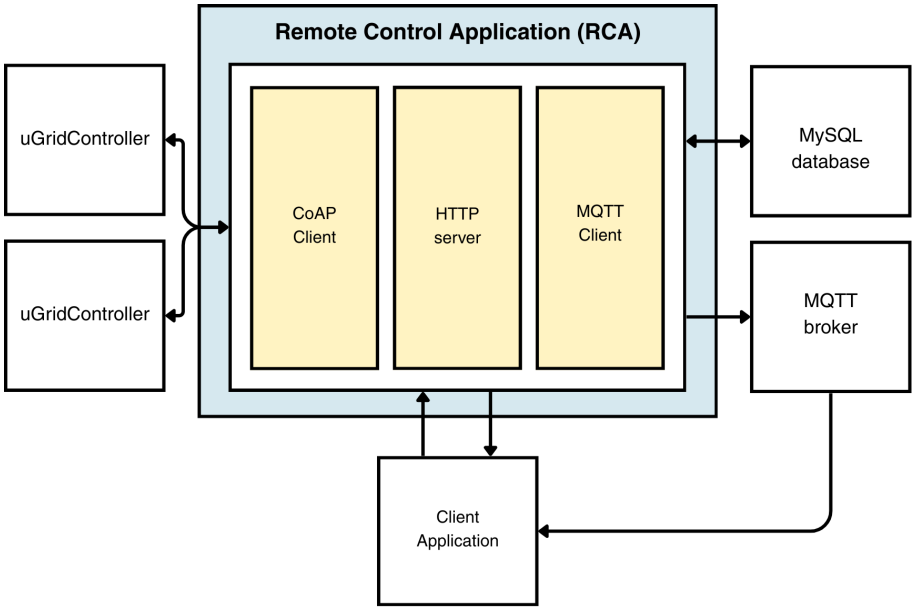

Figure 2.6: Architecture of the RCA. It collects historical data and sends commands to uGridControllers, sends HTTP responses to the Client Application, sends asynchronous communications publishing on an MQTT broker and finally stores data using a MySQL database

The RCA serves four main purposes:

- translate HTTP requests into CoAP requests to the uGridControllers in order to execute commands. Since the RCA is not limited to a resource constrained environment it can execute more computationally intensive protocols such as HTTP
- Parse the incoming telemetry and implement data persistence by saving it into the MySQL database.

- Implement some closed loop control logic, for instance a client can specify a target objective, such as not using the batteries at all, it will be RCA's responsibility to perform this operation.
- Send asynchronous safety-critical communications to the client by publishing on an MQTT broker

2.6 Client Application

The Client Application is the front-end of the overall systems. For educational purposes it consists of a Command Line interface program written in Python that allows the client to visualize real-time data and send commands to the microgrid.

The only messages exchanged by the client application are:

- Synchronous HTTP requests to the RCA for grid state updates
- Asynchronous MQTT messages received by the MQTT for critical warnings

3 Implementation

This chapter details the **implementation** of the BatteryController, the uGridController, the RCA and the Client Application.

3.1 BatteryController

The BatteryController implements both a CoAP server and CoAP client. The former is used to provide a lightweight mechanism for monitoring the current battery state through observable resources, while ensuring Quality of Service (QoS) via the **CON** (Confirmable) mechanism, the latter instead permits registering a new battery pack upon boot time.

The implementation of the BatteryController relies on a Finite State Machine (FSM) made by three main states, a visual representation is depicted in Figure 3.1:

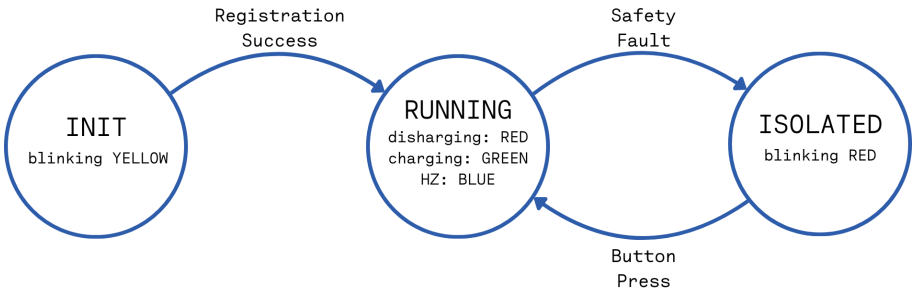

Figure 3.1: FSM representation of BatteryController

- `INIT`: when the system boots up it enters the `INIT` state. In this state the device tries to register to the respective uGridController. For simplicity, the endpoint of the target `uGridController` is hard-coded into the device firmware. This state is visually indicated by a **blinking yellow LED**.
- `RUNNING`: once the device is correctly registered with the appropriate uGridController it enters the `RUNNING` state, signaled when the yellow LED stops blinking. In this state, the device monitors the battery status and stores data in a **queue of 10 states**, enabling the execution of a local Machine Learning (ML) model.

  The primary CoAP resource exposed for monitoring is `/dev/state`, which

accepts only the **GET** method.

While in the `RUNNING` state, the battery operates in one of three possible sub-states:

- **High Impedance**: Signaled by a fixed **blue LED**. In this state, the battery is neither charging nor discharging into the microgrid.
- **Discharging**: Signaled by a fixed **red LED**. The battery is discharging power into the microgrid.
- **Charging**: Signaled by a fixed **green LED**. The battery is absorbing power from the microgrid.

- `ISOLATED`: if the battery requires manual intervention due to low SoH or detected faults, the BatteryController enters the `ISOLATED` state, indicated by a **blinking red LED**. Recovery from this state requires manual intervention; the battery can only return to the `RUNNING` state by **pressing the button** or rebooting the system.

3.1.1 `/dev/state`

```
{
  "V": 395,
  "I": 75,
  "T": 2436,
  "S": 7900,
  "H": 9100,
  "St": 2
}
```

"V": 395, "I": 75, "T": 2436, "S": 7900, "H": 9100, "St": 2

The current state of the battery is made available via a **GET** request on `/dev/state` resource.

Given the small size of the response body, $(20 - 30)$ bytes, data is transmitted in JSON format. To ensure a lightweight and fast implementation, the message is constructed using `snprintf` and parsed exploiting the parsing capabilities of `sscanf` function.

3.1.2 `/dev/power`

Request:

```
{
    "u": 1200
}
```

"u": 1200

Response:

```
{
    "u": 1000
}
```

"u": 1000

Actuation of the battery's power relay is managed through the `/dev/power` resource. Using the **PUT** method, the uGridController can regulate the power flow: positive values indicate charging, zero sets the battery to high impedance, and negative values indicate discharging. Any value set by the uGridController may be overwritten by the safety policies of the BatteryController for instance due to power limits or other critical fault, the actual value will be returned in the response.

Given the small size of both the request and the response body, $(10 - 15)$ bytes, data is transmitted in JSON format. As for the `/dev/state` message, the `/dev/power` is constructed and parsed using `snprintf` and `sscanf` functions.

3.2 uGridController

The uGridController implements both a CoAP server and CoAP client. The former is used to provide a lightweight mechanism for observing the state of registered BatteryControllers, while ensuring a QoS via the **CON** mechanism and registration of new battery packs within the application, the latter instead permits the actuation of power relays at the BatteryControllers. The operational state of the uGridController is signaled via a **fixed Green LED**.

The core responsibility of the uGridController is to schedule the optimal power flow. This process is triggered periodically (at a frequency `FREQ_COMPUTING`. The control loop consists of three steps:

1. **Prediction:** The internal ML model estimates future PV and load powers from current load, PV power, battery states and sensor values.
2. **Optimization:** A quadratic optimization problem is solved using a lightweight **Projected Gradient** algorithm to determine the ideal power flows with all

the registered batteries.

3. **Actuation:** The computed power commands are sent sequentially to all registered batteries via a CoAP **PUT** request to their respective `/dev/power` resources.

   Since physical constraints may prevent a battery from delivering the exact requested optimal power u^*, each device replies with the actual power u' it can provide. The uGridController calculates the difference between the optimal and actual values and compensates with the power request to the main grid.

The execution of the control loop is signaled via a **fixed blue LED**.

3.2.1 `/dev/state`

```
{
  "count": 2,
  "load": 123,
  "pv": -45,
  "bats": [
    [0, 395, 79, 120, 398, 75, 2436, 91, 2],
    [3, 400, 81, 110, 401, 70, 2410, 90, 1]
  ]
}
```

"count": 2, "load": 123, "pv": -45, "bats": [[0, 395, 79, 120, 398, 75, 2436, 91, 2], [3, 400, 81, 110, 401, 70, 2410, 90, 1]]

As depicted in Figure 2.5, the uGridController serves as the primary interface between the RCA and the WSN. Since the uGridController must continuously observe the status of all registered batteries to schedule optimal power flows, it already possesses the current state of the entire local cluster. For this reason, the RCA queries for the state of the batteries directly at the uGridController, rather than polling individual nodes.

In this sense, the uGridController acts as a reverse proxy for the BatteryControllers significantly reducing traffic overhead on the WSN. To ensure consistency, the uGridController exposes the same **GET** on `/dev/state` resource found on the individual BatteryControllers **but with a different format**, the maximum number of batteries that a single uGridController can manage is limited to 5.

Given that the size of the response could reach a significant amount of bytes as the number of controlled batteries grows, data is transmitted in CBOR format using the predefined CBOR module of Contiki-NG.

3.2.2 /dev/register

```
{
  "battery-id": 3
}
```

"battery-id": 3

The `/dev/register` resource is exposed by the uGridController to allow BatteryControllers to dynamically join the local uGrid cluster. Each battery node issues a **POST** request to this endpoint during its INIT phase to join the uGrid.

Unlike other resources, the registration request carries a minimal payload consisting of the `battery_id` encoded as a plain integer. The controller retrieves the IPv6 source address directly from the request's metadata.

If the maximum number of batteries is not exceeded, the uGridController replies with a `2.01 Created` CoAP response code. Instead, if the registry fails, the request is rejected with `5.03 Service Unavailable`.

3.2.3 /ctrl/mpc

```
{
  "a": 125,
  "b": 75,
  "g": 50,
  "p": 20
}
```

"a": 125, "b": 75, "g": 50, "p": 20

As anticipated in Chapter 2, the MPC optimization params can be dynamically changed via a **PUT** on the `/ctrl/mpc` resource. Given that the size of the request is small, $(10 - 15)$ bytes, data is transmitted in JSON format. To ensure a lightweight and fast implementation, the message is constructed using `snprintf` and parsed exploiting the parsing capabilities of `sscanf` function.

3.2.4 /ctrl/obj


```
{
  "idx": 3,
  "power": 125,
  "clear": 0
}
```

"idx": 3, "power": 125, "clear": 0

Objectives are sent with a **PUT** on the `/ctrl/obj` resource. Given the small size of the request, $(10 - 15)$ bytes, data is transmitted in JSON format. To ensure a lightweight and fast implementation, the message is constructed using `snprintf` and parsed exploiting the parsing capabilities of `sscanf` function.

3.3 Remote Control Application

The RCA, exposes a small HTTP REST interface via Flask and publishes alert notifications via MQTT. Since the RCA layer is not the main focus of this academic project, this section only lists the endpoints,and MQTT topics together with their primary purpose.

- **`GET /api/status`**: Returns the latest status received from the uGridController.
- **`POST /api/batteries/{ugrid_id}/{bat_idx}/objective`**: Receives an objective command for a specific battery.
- **`DELETE /api/batteries/{ugrid_id}/{bat_idx}/objective`**: Clears a previously sent objective for a specific battery.
- **`GET /api/batteries/{ugrid_id}/{bat_idx}/history?limit=N`**: Retrieves telemetry samples for the selected battery from the database.
- **`GET /api/ugrids/{ugrid_id}/mpc_params`**: Returns the latest MPC parameters stored in the database for the selected uGrid.
- **`POST /api/ugrids/{ugrid_id}/mpc_params`**: Updates MPC parameters for the selected uGridController.
- **`GET /api/alerts?limit=N`**: Returns the latest alerts stored in the database.
- **MQTT Alerts**: The RCA publishes alerts to the MQTT broker. Alerts are published under the topic `ugrid/alerts/<level>/<ugrid_id>[/<battery_index>]`, where `<level>` is the alert severity.

3.4 Client Application

The CA provides an interactive CLI application for monitoring and controlling the LBSS system. It periodically retrieves the system state from the Remote Control Application (RCA) via HTTP and displays aggregated information about uGrids, batteries, power flows, and economic indicators.

The CA also subscribes to the MQTT alert topics published by the RCA, allowing asynchronous visualization of warnings and critical events. In addition, the CLI enables the user to issue commands such as setting battery objectives or updating MPC parameters.

Figure 3.2 shows two screenshots of the Client Application, highlighting both the monitoring dashboard and the alert panel.

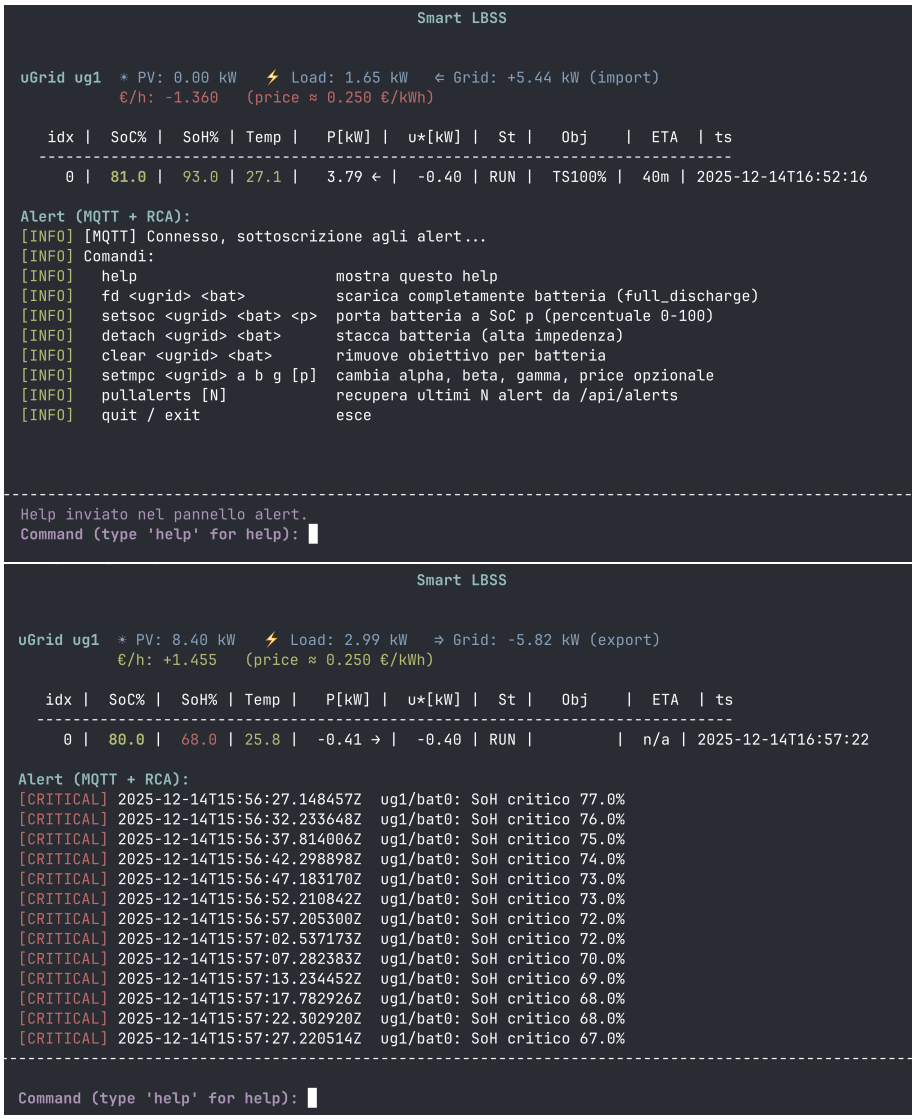

Figure 3.2: CA CLI showing real-time uGrid status, battery metrics, power flows, and MQTT/RCA alerts.