Maëlle Gournay

28.08.2025

Technical Assignment: OPC UA Temperature Monitoring System

Goal: Build a temperature monitoring application that connects to an OPC UA server, reads sensor data, processes alarms based on JSON configuration, and publishes results back to the server.

Content

This project is a Python-based monitoring tool for reading sensor values from an OPC UA server (tested with *Prosys OPC UA Simulation Server – Free Version*). It supports real-time monitoring, alarm management, trend detection, historization, and a web dashboard.

It features:

- OPC UA Connection to Prosys Simulation Server (anonymous login).
- Real-time monitoring of multiple sensors.
- Alarm management (High, High-High, Low, Low-Low).
- Trend detection (rapid rise or fall).
- Alarm historization (JSON and CSV).
- Interactive console menu for simulation control.
- Web dashboard (Flask and SocketIO) for live updates.
- Dynamic configuration reload without restart.
- Shutdown and data export on exit.

System workflow:

1. Establish OPC UA connection.
2. Start sensor monitoring with subscriptions.
3. Detect and manage alarms.
4. Detect trends (rapid value changes).
5. Show real-time updates on the web dashboard.
6. Export alarm history (JSON).
7. Stop monitoring and disconnect safely.

Requirement:

```
python3 -m pip install opcua flask flask-socketio keyboard
```

To run the program on the monitor:

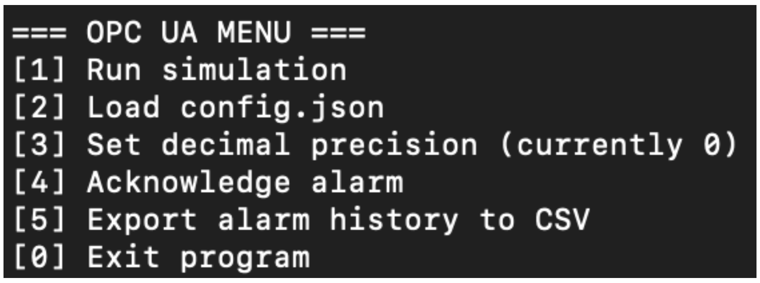
```
python3 opcua_monitor.py
```

Directory structure:

```
├── opcua_monitor.py       # Main Python script
├── config.json            # Configuration file
├── templates/
│   └── index.html         # Web dashboard
├── alarm_history_*.json   # Auto-saved alarm history
├── alarm_history_*.csv    # Exported alarm history
```

1. Menu – CLI interface

When the code is launched on the monitor, the configuration file (config.json) is loaded and a menu is displayed:



```
=== OPC UA MENU ===
[1] Run simulation
[2] Load config.json
[3] Set decimal precision (currently 0)
[4] Acknowledge alarm
[5] Export alarm history to CSV
[0] Exit program
```

Option	Description
Run simulation	Start the monitoring with alarm and trend detection.
Load the configuration file	The configuration file can be reloaded.
Set decimal precision (currently n)	Adjust the output format of the temperature values, indicating the current number of decimals.
Acknowledge alarm	Mark the alarms as acknowledged (locally if the server doesn't support it).
Export alarm history to CSV	Save the alarms as a CSV file, with the same format as the JSON file.
Exit program	Stop the monitoring and exit the program.

The whole menu is handled by the function `menu()` works as a CLI interface and uses the Boolean variable `exit_program` and the variable `DECIMALS`.

The menu uses an if / elif loop to go through the different options of the menu and print "Invalid choice" if the value entered by the user is not defined.

The variable `exit_program` is used when the exit program option is entered: the variable goes from wrong to true.

The variable `DECIMALS` is used to replace n in the name of the third option (Set decimal precision (currently n)) to know how much decimals on the temperature values are going to be displayed during the simulation. This number can be changed without exiting the program by just going back to the menu.

2. Run simulation

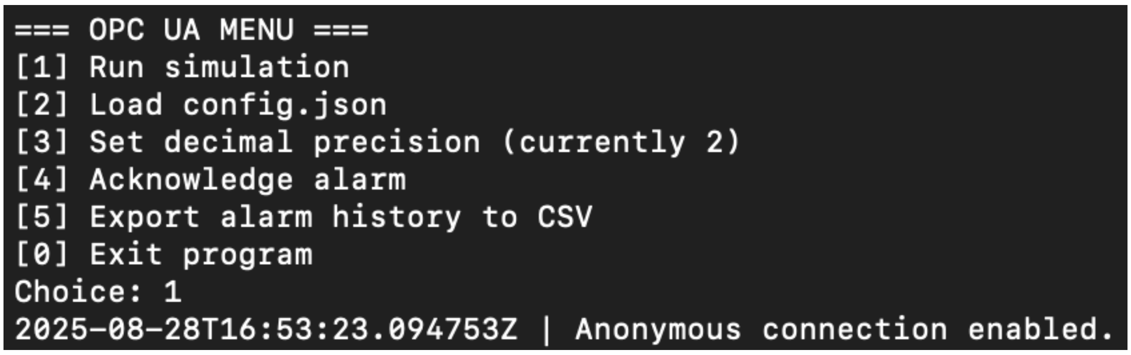
a. OPC UA server connection

To connect to the server, the function `connect_to_opcua()` is used.

Here the variable `ENDPOINT` is used with the following link: "opc.tcp://localhost:53530/OPCUA/SimulationServer". If the server is open and the server status is: Running, then the function can connect itself to the server with an anonymous connection.

A connection with a username and a password is only possible with the paid version (the free version doesn't allow to write on the server, only read).

The connection uses a retry logic with a maximum number of attempts `MAX_RETRIES` defined in the program. The different attempts occur every `RETRY_DELAY` period.



```
=== OPC UA MENU ===
[1] Run simulation
[2] Load config.json
[3] Set decimal precision (currently 2)
[4] Acknowledge alarm
[5] Export alarm history to CSV
[0] Exit program
Choice: 1
2025-08-28T16:53:23.094753Z | Anonymous connection enabled.
```

b. Utility functions

1. Timestamp ISO 8601

In order to have the timestamps with the ISO 8601format, the function `iso(ts: datetime) -> str` is used. It will replace the timestamp format with the UTC format.

2. Convert the numbers to float

To convert the values to float when possible (to normalize the datas), the function `normalize_number(val)` is used.

It tries to convert a Boolean value into a float, returning 1.0 for True, 0.0 for False, or the float equivalent of numbers and numeric strings. It will return None if the conversion fails.

c. Alarm detection

The functions used for the alarm detections are:

- `emit_event(event_type, sensor, level, value, threshold, now_ts, started_at=None)`

This function manages alarm events: when an alarm is raised (`ALARM_ACTIVE`) it logs a new alarm entry with details from the payload. Those informations will also go in the dictionary `alarm_history`. When an alarm is cleared (`ALARM_CLEAR`) it updates the matching active alarm with its duration and marks it inactive. It also updates statistics and broadcasts the event via `socketio`.

Example:

```
alarm_history = {
  "alarms": [   # chronological list of alarms
    {
      "timestamp": str,        # ISO 8601
      "sensor": str,           # Sensor name
      "nodeId": str,           # OPC UA NodeId
      "type": str,             # HH / H / L / LL
      "priority": int,         # severity ranking
      "value": str,            # formatted measurement
      "threshold": float,      # triggering threshold
      "duration": float|null,  # sec (only set on clear)
      "active": bool,          # alarm status
      "acknowledged": bool     # local acknowledgment
    }
  ],
  "statistics": {
    "total_alarms": int,
    "active_alarms": int,
    "by_type": {"HH":int,"H":int,"L":int,"LL":int}
  }
}
```

- `check_levels(sensor, value, now_ts)`

This function checks a sensor's value against its configured alarm thresholds and triggers or clears alarms accordingly. It uses the global variable `TIME_DELAY` and the dictionary `states` to have the different threshold levels.

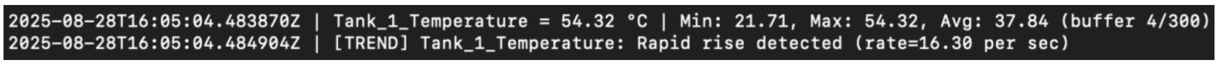
For each threshold level (H, HH, L, LL), it applies hysteresis (using the deadband to prevent rapid toggling) and requires the condition to hold for a set delay (`TIME_DELAY`) before marking the alarm active; when conditions return to normal, it clears the alarm and records its duration.

d. Trend detection

```
trend_settings = {
  "rise_rate": float,   # threshold for rising slope (value/sec)
  "fall_rate": float    # threshold for falling slope (value/sec)
}
```

The function is `check_trend(sensor, value, now_ts)` and it uses the variables `last_values` (dictionary), `trend_settings` (dictionary of the fall and rise rates) and `prev_values` (dictionary).

It skips all non-numeric values and then compares the last value with the previous value and compares the difference with the rates. If the difference is over the rate, then a trend is detected and a message will appear on the terminal with the rate per second.



```
2025-08-28T16:05:04.483870Z | Tank_1_Temperature = 54.32 °C | Min: 21.71, Max: 54.32, Avg: 37.84 (buffer 4/300)
2025-08-28T16:05:04.484904Z | [TREND] Tank_1_Temperature: Rapid rise detected (rate=16.30 per sec)
```

e. Subscription handler

The class method `Subhandle` handles incoming data changes from sensors in real time with the function `datachange_notification(self, node, val, data)`. The global variable `stop_monitoring` and the dictionary `buffers` are used.

For each new value:

1. It looks up the corresponding sensor.
2. Determines the timestamp (`src_ts` - SourceTimestamp if available, otherwise current time).
3. Converts the value to a number with `normalize_number(val)` and appends it to a buffer in the dictionary.
4. Calculates and prints min, max, and average values from the buffer.
5. Updates `last_values` and `prev_values` for real-time tracking.
6. Emits the updated values to clients via `socketio`.
7. Calls `check_trend(sensor, value, now_ts)` and `check_levels(sensor, value, now_ts)` to monitor trends and trigger/clear alarms.

It also logs warnings for invalid data and gracefully skips processing if monitoring is stopped.

It's essentially the bridge between raw sensor data and the alarm/trending system.

f. Run the simulation

The monitoring loop for the OPC UA sensors is `run_simulation()`.

This program loads the configuration if no sensors are detected and prepares the JSON alarm history file before connecting to the OPC UA server.

Then the function `print_initial_values(client)` reads and displays each sensor's starting values among with the minimum, maximum and average value.

A subscription for all sensor is created: `SubHandler.datachange_notification` receives updates in real time.

A while not loop checks if the Boolean variable stop_monitoring is wrong, permitting the updates of the temperatures. If “q” and ENTER are pressed, stop_monitoring becomes true.

In the loop, the server is periodically ping (node i=1008 – node from my first sensor) to detect if there is a disconnection.

g. Error handling

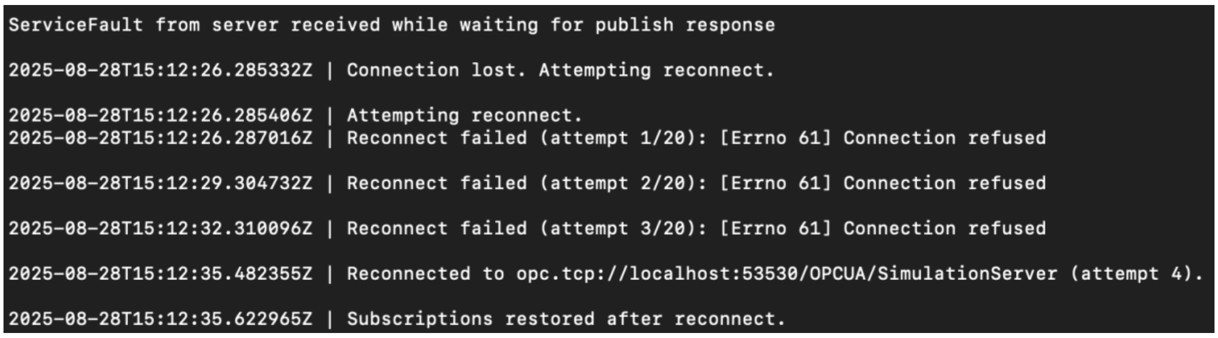
1. Keyboard handling.

MacOS does not support `keyboard.is_pressed("q")`, so I had to use `input().strip().lower() == "q"` and ENTER must be pressed.

2. Connection loss recovery.

Automatic reconnection attempts with the function `reconnect_client()`.

Because `keyboard.is_pressed("q")` doesn’t work with MacOS, q must be pressed followed by ENTER. That means the program is going to wait at the ServiceFault step until ENTER is pressed. Afterwards, the reconnect attemps will begin.



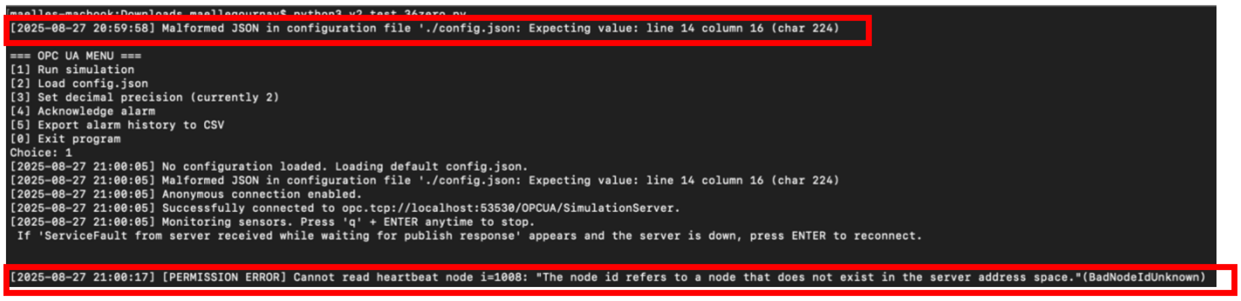
```
ServiceFault from server received while waiting for publish response

2025-08-28T15:12:26.285332Z | Connection lost. Attempting reconnect.

2025-08-28T15:12:26.285406Z | Attempting reconnect.
2025-08-28T15:12:26.287016Z | Reconnect failed (attempt 1/20): [Errno 61] Connection refused

2025-08-28T15:12:29.304732Z | Reconnect failed (attempt 2/20): [Errno 61] Connection refused

2025-08-28T15:12:32.310096Z | Reconnect failed (attempt 3/20): [Errno 61] Connection refused

2025-08-28T15:12:35.482355Z | Reconnected to opc.tcp://localhost:53530/OPCUA/SimulationServer (attempt 4).

2025-08-28T15:12:35.622965Z | Subscriptions restored after reconnect.
```

Subscriptions are then restored, and the initials values will be read again.

3. Invalid data type handling.

Ignore and log invalid values.

I can't inject bad data with good JSON, which is why I end up with:



```
[2025-08-27 20:59:58] Malformed JSON in configuration file './config.json: Expecting value: line 14 column 16 (char 224)

=== OPC UA MENU ===
[1] Run simulation
[2] Load config.json
[3] Set decimal precision (currently 2)
[4] Acknowledge alarm
[5] Export alarm history to CSV
[0] Exit program
Choice: 1
[2025-08-27 21:00:05] No configuration loaded. Loading default config.json.
[2025-08-27 21:00:05] Malformed JSON in configuration file './config.json: Expecting value: line 14 column 16 (char 224)
[2025-08-27 21:00:05] Anonymous connection enabled.
[2025-08-27 21:00:05] Successfully connected to opc.tcp://localhost:53530/OPCUA/SimulationServer.
[2025-08-27 21:00:05] Monitoring sensors. Press 'q' + ENTER anytime to stop.
 If 'ServiceFault from server received while waiting for publish response' appears and the server is down, press ENTER to reconnect.

[2025-08-27 21:00:17] [PERMISSION ERROR] Cannot read heartbeat node i=1008: "The node id refers to a node that does not exist in the server address space."(BadNodeIdUnknown)
```

Since no valid alarm_settings got loaded, the code kept falling back to the hardcoded i=1008 check.

The node 1008 is the node of the first analyzed sensor.

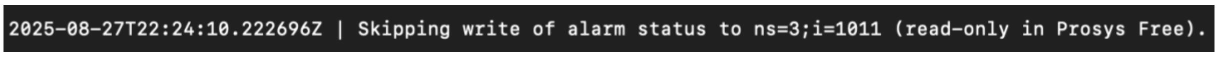
In the Prosys free server, if the simulation model is loaded, i=1008 does exist. But because the configuration failed, the setup was incomplete and BadNodeIdUnknown was hit.

4. Malformed JSON configuration.

Validation with error messages before the menu.

5. Permission issues.

Handle restricted OPC UA nodes gracefully. Because the OPC UA simulation server is only the free version, writing status on nodes is not possible. Therefore, the alarm status are not written, but this message comes on the terminal.



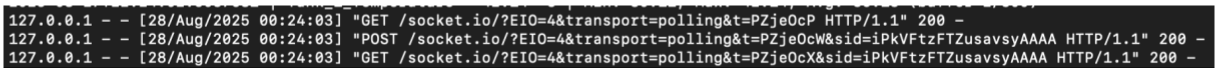
```
2025-08-27T22:24:10.222696Z | Skipping write of alarm status to ns=3;i=1011 (read-only in Prosys Free).
```

h. Web dashboard

The system includes an HTML and a SocketIO dashboard to visualize the alarms in a real-time monitoring.

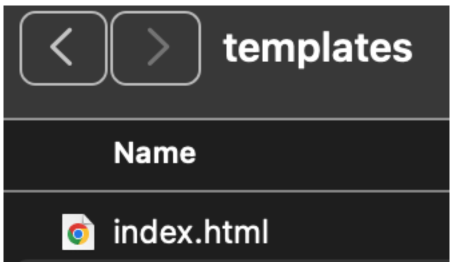
Flask and SocketIO are used:

- Flask serves the web app.



```
127.0.0.1 - - [28/Aug/2025 00:24:03] "GET /socket.io/?EIO=4&transport=polling&t=PZjeOcP HTTP/1.1" 200 -
127.0.0.1 - - [28/Aug/2025 00:24:03] "POST /socket.io/?EIO=4&transport=polling&t=PZjeOcW&sid=iPkVFtzFTZusavsyAAAA HTTP/1.1" 200 -
127.0.0.1 - - [28/Aug/2025 00:24:03] "GET /socket.io/?EIO=4&transport=polling&t=PZjeOcX&sid=iPkVFtzFTZusavsyAAAA HTTP/1.1" 200 -
```

- SocketIO pushes the live updates from the python code.



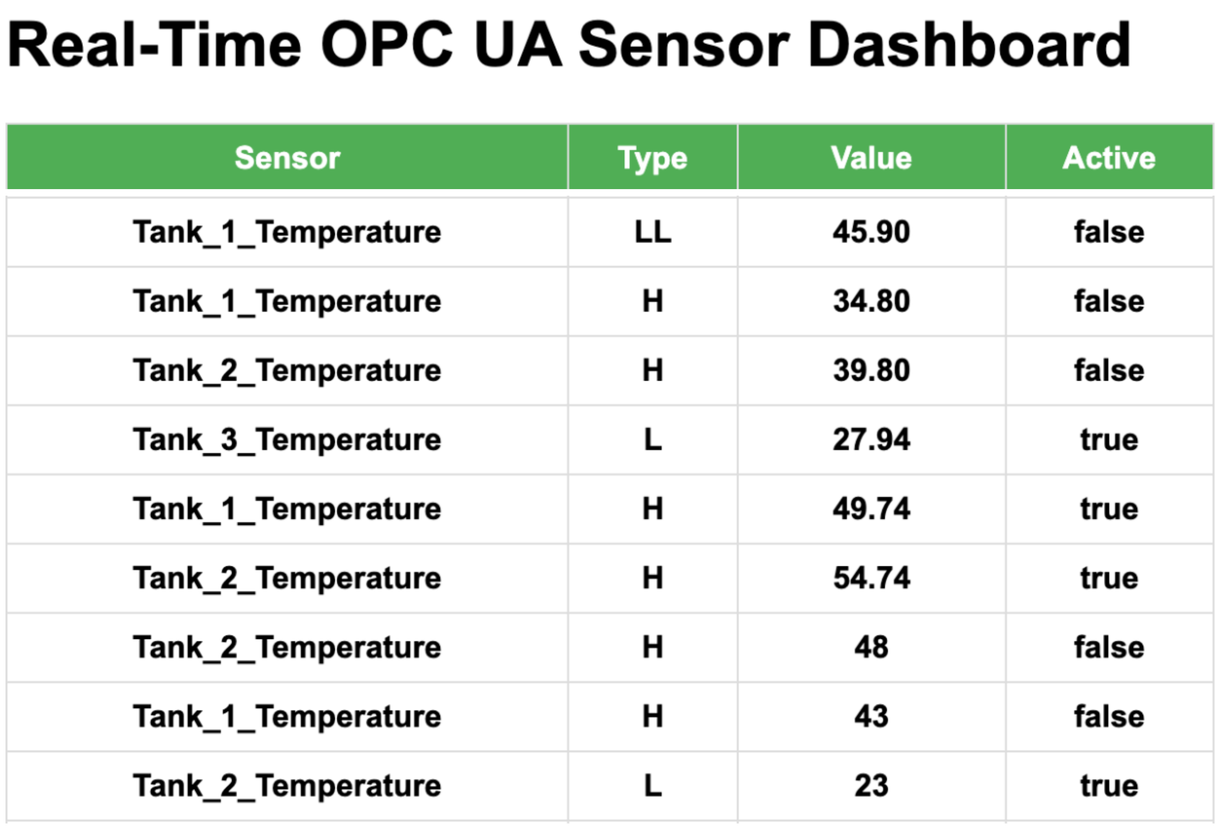
For the dashboard on the web, a, html code is needed. This code must be named index.html and be placed in the order templates (lower cases). This order must be placed in the same order as the python file.

The HTML program defines a dashboard that displays incoming alarm events from a server. It includes SocketIO (`socket.io.min.js`) to receive live events from the python program.

The script connects to the server via SocketIO: `const socket = io()` and listens for alarm events:

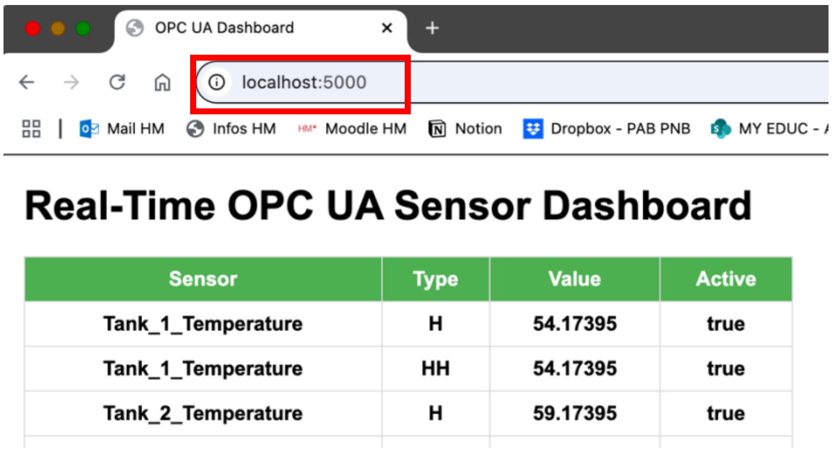
```
socket.on('alarm', (data) => { ... })
```

Each new alarm received is logged to the console and appended as a new row in the table with sensor name, alarm type, measured value, and active status.



Real-Time OPC UA Sensor Dashboard

Sensor	Type	Value	Active
Tank_1_Temperature	LL	45.90	false
Tank_1_Temperature	H	34.80	false
Tank_2_Temperature	H	39.80	false
Tank_3_Temperature	L	27.94	true
Tank_1_Temperature	H	49.74	true
Tank_2_Temperature	H	54.74	true
Tank_2_Temperature	H	48	false
Tank_1_Temperature	H	43	false
Tank_2_Temperature	L	23	true



Real-Time OPC UA Sensor Dashboard

Sensor	Type	Value	Active
Tank_1_Temperature	H	54.17395	true
Tank_1_Temperature	HH	54.17395	true
Tank_2_Temperature	H	59.17395	true

The web page is located on the port 5000 of the localhost. To access it, just write localhost:5000 in the url search.

To have this dashboard, some functions and imports are needed in the code:

```
from flask import Flask, render_template

from flask_socketio import SocketIO


# Flask & SocketIO setup

app = Flask(__name__)

socketio = SocketIO(app, cors_allowed_origins="*",
async_mode="threading")


@app.route("/")

def index():

    return render_template("index.html")
```

To write the data on the dashboard, this line is needed:

```
socketio.emit("alarm", payload)
```

In the code, the socketio.emit is written in the following functions:

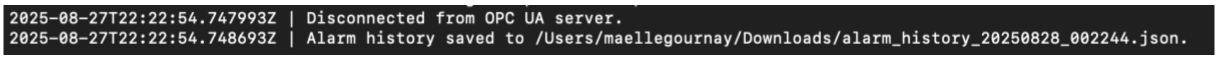
- `emit_event(event_type, sensor, level, value, threshold, now_ts, started_at=None)`
- `SubHandler datachange_notification(self, node, val, data)`

At the beginning of the program, the connection is also made with the following line:

```
threading.Thread(target=lambda: socketio.run(app,
host="127.0.0.1", port=5000), daemon=True).start()
```

i. Disconnect

When q and ENTER are pressed, the program will disconnect itself from the OPC UA server: `client.disconnect()`

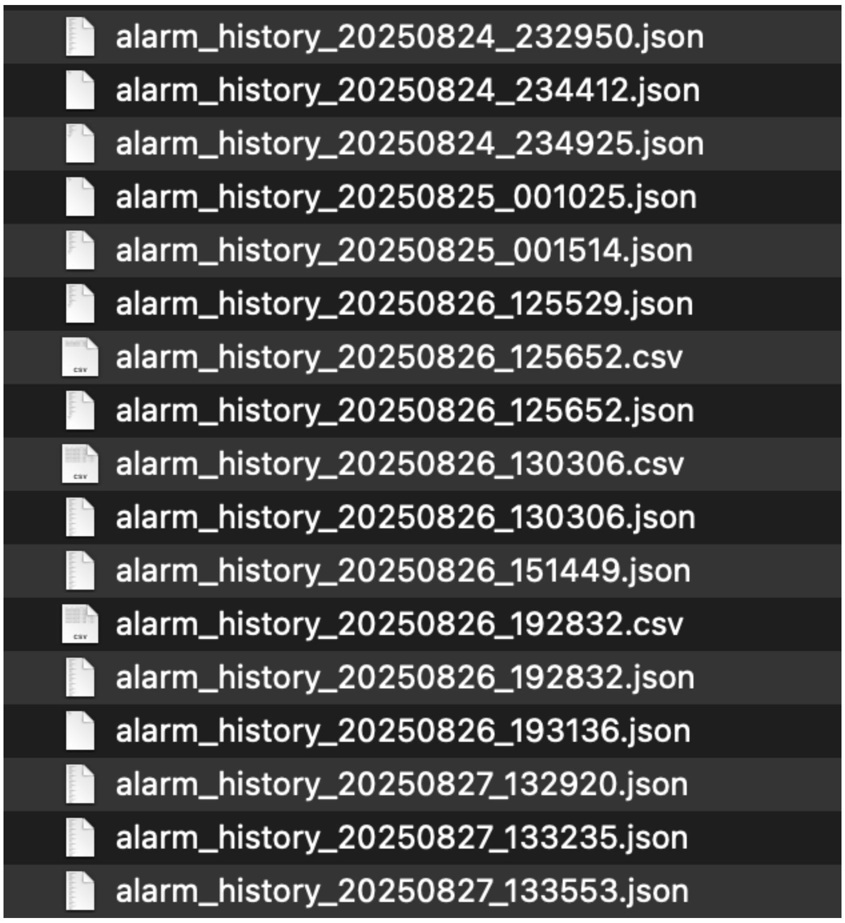


```
2025-08-27T22:22:54.747993Z | Disconnected from OPC UA server.
2025-08-27T22:22:54.748693Z | Alarm history saved to /Users/maellegournay/Downloads/alarm_history_20250828_002244.json.
```

Afterwards, the alarm history will be saved (name: alarm_history_timestamp.json). The files can be found in the same order as the order from the python program.

The function is called `write_alarm_history()` and uses the file path of the program and a variable called `alarm_history` as dictionary.

Example:



```
alarm_history_20250824_232950.json
alarm_history_20250824_234412.json
alarm_history_20250824_234925.json
alarm_history_20250825_001025.json
alarm_history_20250825_001514.json
alarm_history_20250826_125529.json
alarm_history_20250826_125652.csv
alarm_history_20250826_125652.json
alarm_history_20250826_130306.csv
alarm_history_20250826_130306.json
alarm_history_20250826_151449.json
alarm_history_20250826_192832.csv
alarm_history_20250826_192832.json
alarm_history_20250826_193136.json
alarm_history_20250827_132920.json
alarm_history_20250827_133235.json
alarm_history_20250827_133553.json
```

3. Load the configuration file

The program uses a configuration JSON file for sensors, alarms and trend settings.

There is a list of the monitored sensors (`nodeID`, `name`, thresholds and `scanRate`), the alarm settings (acknowledgement requirement, delay and notification node) and the trend settings (thresholds to detect rapid rise / fall).

Example:

```
{
  "sensors": [
    {
      "nodeId": "ns=3;i=1008",
      "name": "Tank_1_Temperature",
      "unit": "°C",
      "alarms": {
        "high_high": 50,
        "high": 45,
        "low": 25,
        "low_low": 20
      },
      "deadband": 2.0,
      "scanRate": 1000
    }
  ],
  "alarmSettings": {
    "requireAcknowledgment": true,
    "timeDelay": 0,
    "notificationNode": "ns=3;i=1011"
  },
"trendSettings": {
    "riseRate": 1.0,
    "fallRate": -1.0
  }
}
```

The function used to load and initialize the configuration is called `load_config_json()` and uses the list `sensor`, the variable `TIME_DELAY` and the dictionaries `alarm_settings`, `buffers`, `states`, `alarm_history` and `trend_settings`.

It opens the file `./config.json` and parses it, except when the file is malformed, or the file is missing. It then extracts the list of sensors, and they are placed in the variable `sensor`. The global settings are also loaded: `alarmSettings` in the dictionary `alarm_settings` and `trendSettings` in the dictionary `trend_settings`.

Afterwards the sensor scan rates are clamped to ensure that each sensor's `scanRate` is within the allowed range (`MIN_SCANRATE` – `MAX_SCANRATE`). If the values are clamped, warnings and the new scanrates are written.

The dictionary `buffers` is then initialized: there are rolling deques for each sensor to store recent sample (size based on five minutes of data). The dictionary `states` is also updated for each sensor and alarm level (HH, H, L, LL).

The dictionary `alarm_history` is cleared and alarm tracking are resetted (including statistics for totals, active alarms, and counts by type of alarm).

4. Set decimal precision (currently n)

The function `fmt_num(x)` gives the good number of decimals for the temperature measures.

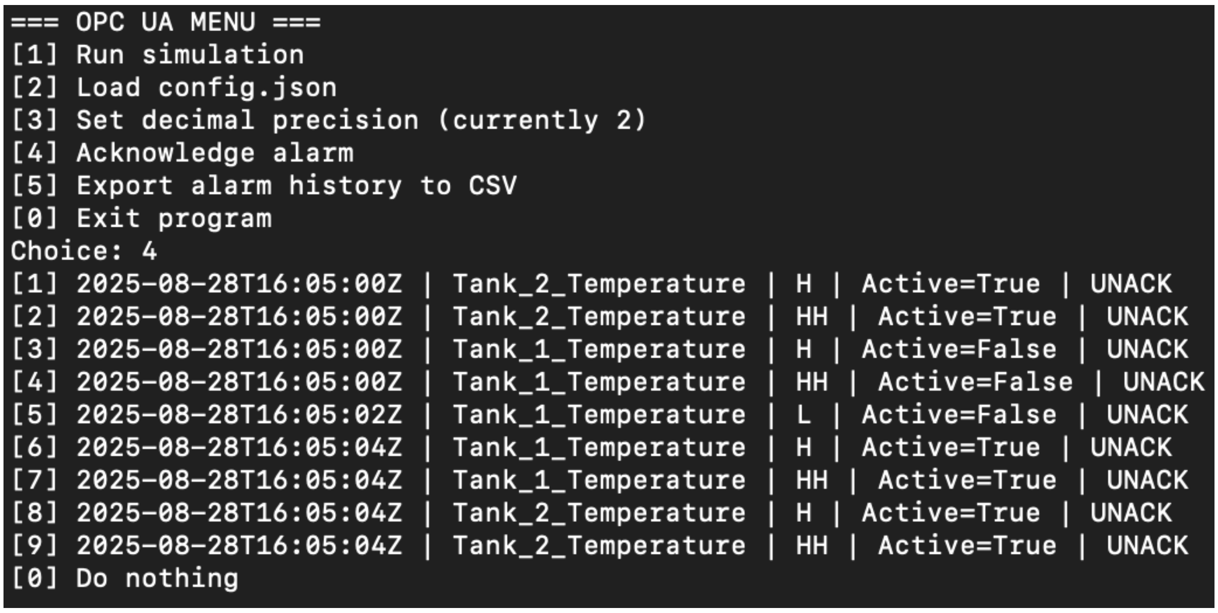
`DECIMALS` is a global variable in the program that is used also in the menu:

```
elif choice_int == 3:
    try:
        new_dec = int(input("Enter number of decimals (e.g.
    2):").strip())
        if new_dec >= 0:
            DECIMALS = new_dec
            print(f"Decimal precision set to {DECIMALS}.")
        else:
            print("Please enter a non-negative integer.")
    except ValueError:
        print("Invalid number.")
```

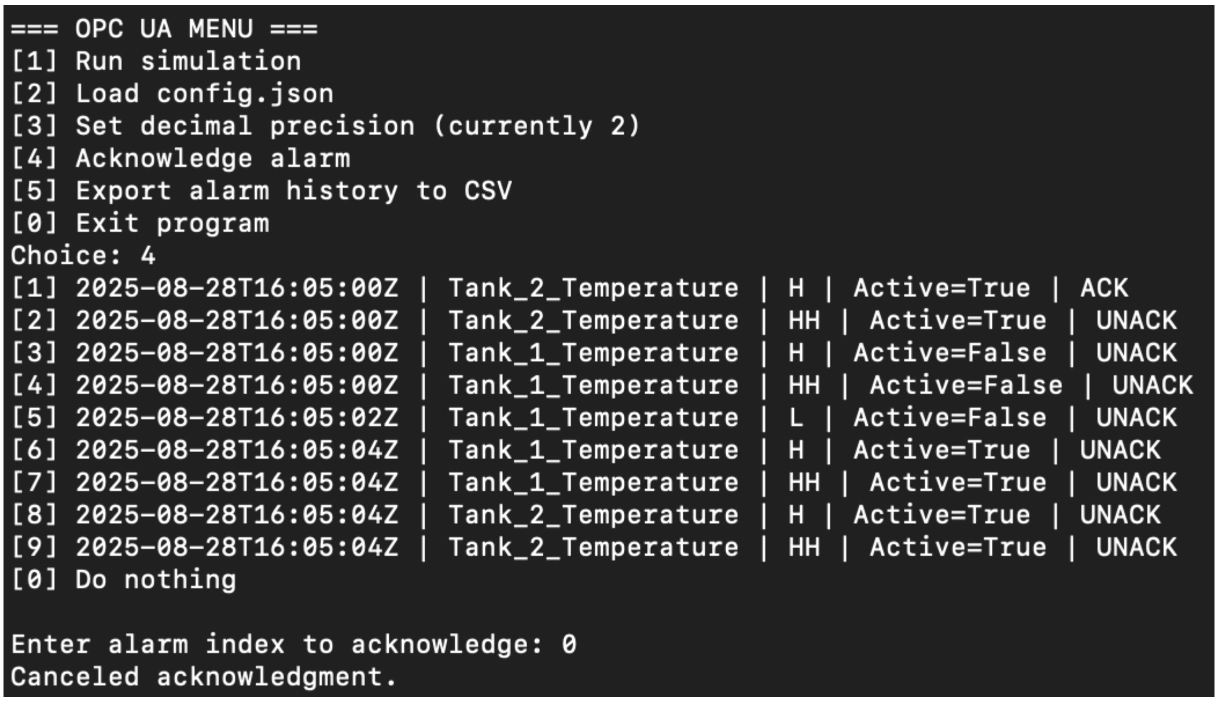
5. Acknowledge alarm

The function is called `acknowledge_alarm(alarm_index, client=None, comment="Acknowledge via client”)`. Here will the alarm history be parsed.

A method call on the OPC UA is here written, in the case the server is not the free version. Here, it doesn’t work. The history will be updated locally (on the JSON and CSV files).



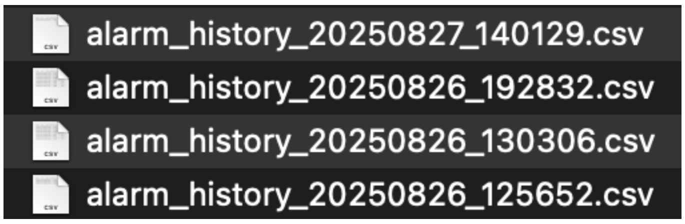
```
=== OPC UA MENU ===
[1] Run simulation
[2] Load config.json
[3] Set decimal precision (currently 2)
[4] Acknowledge alarm
[5] Export alarm history to CSV
[0] Exit program
Choice: 4
[1] 2025-08-28T16:05:00Z | Tank_2_Temperature | H | Active=True | UNACK
[2] 2025-08-28T16:05:00Z | Tank_2_Temperature | HH | Active=True | UNACK
[3] 2025-08-28T16:05:00Z | Tank_1_Temperature | H | Active=False | UNACK
[4] 2025-08-28T16:05:00Z | Tank_1_Temperature | HH | Active=False | UNACK
[5] 2025-08-28T16:05:02Z | Tank_1_Temperature | L | Active=False | UNACK
[6] 2025-08-28T16:05:04Z | Tank_1_Temperature | H | Active=True | UNACK
[7] 2025-08-28T16:05:04Z | Tank_1_Temperature | HH | Active=True | UNACK
[8] 2025-08-28T16:05:04Z | Tank_2_Temperature | H | Active=True | UNACK
[9] 2025-08-28T16:05:04Z | Tank_2_Temperature | HH | Active=True | UNACK
[0] Do nothing
```



```
=== OPC UA MENU ===
[1] Run simulation
[2] Load config.json
[3] Set decimal precision (currently 2)
[4] Acknowledge alarm
[5] Export alarm history to CSV
[0] Exit program
Choice: 4
[1] 2025-08-28T16:05:00Z | Tank_2_Temperature | H | Active=True | ACK
[2] 2025-08-28T16:05:00Z | Tank_2_Temperature | HH | Active=True | UNACK
[3] 2025-08-28T16:05:00Z | Tank_1_Temperature | H | Active=False | UNACK
[4] 2025-08-28T16:05:00Z | Tank_1_Temperature | HH | Active=False | UNACK
[5] 2025-08-28T16:05:02Z | Tank_1_Temperature | L | Active=False | UNACK
[6] 2025-08-28T16:05:04Z | Tank_1_Temperature | H | Active=True | UNACK
[7] 2025-08-28T16:05:04Z | Tank_1_Temperature | HH | Active=True | UNACK
[8] 2025-08-28T16:05:04Z | Tank_2_Temperature | H | Active=True | UNACK
[9] 2025-08-28T16:05:04Z | Tank_2_Temperature | HH | Active=True | UNACK
[0] Do nothing

Enter alarm index to acknowledge: 0
Canceled acknowledgment.
```

6. Export alarm history to CSV

The alarm history will be saved (name: alarm_history_timestamp.csv) under the same name as the JSON file, only with a CSV format instead. The files can be found in the same order as the order from the python program and the JSON files.



```
alarm_history_20250827_140129.csv
alarm_history_20250826_192832.csv
alarm_history_20250826_130306.csv
alarm_history_20250826_125652.csv
```

The function is here called `export_alarm_history_csv()` and uses also the file path of the program and a variable called `alarm_history` as dictionary.

7. Exit program

The boolean global variable `exit_program` is always on false, except when 0 is pressed in the menu.

In the menu function:

```
elif choice_int == 0:
            print("Exiting program.")
            exit_program = True
```