INNOVATIVE SMART SYSTEMS

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Middleware for IoT

oneM2M standard

5A ISS
2024-2025

Noël Jumin
Clément Gauché

Noël Jumin
Clément Gauché

Noël Jumin
Clément Gauché

Introduction

This report focuses on implementing a middleware for IoT systems using the oneM2M standard. The experiment uses the ACME oneM2M Common Service Entity, a lightweight Python-based implementation of oneM2M. Working through this lab aimed to understand oneM2M service layers, manipulate resources, and simulate device behavior.

Noël Jumin
Clément Gauché

Installation and Configuration of ACME

To install ACME, we followed the instructions from the tutorial below:
https://acmecse.net/setup/Installation/#__tabbed_1_2

The ACME CSE was set up on a local machine to act as an IN-CSE (Infrastructure Node Common Service Entity). The process involved to have an environment setted up, before installing all the required dependencies.

To do that we used the following commands:

```
python3 -m venv venv # Setup the environment
source venv/bin/activate # Activate it (on Linux/Mac)
.\venv\Scripts\Activate.ps1 # On Windows
```

Then simply git clone next to this venv folder (ACME github repository).
You'll see (venv) appear on the left of the command line.
You can then launch IN-CSE with the following command:

```
acmecse
```

Or by using the dedicated script:

```
python .\start-cse.py
```

In chapters 1 and 2 of the notebook we saw how to create an AE, a container and an instance as well as how to retrieve data from this.
We can observe the ACME interface from this address: http://localhost:8080/

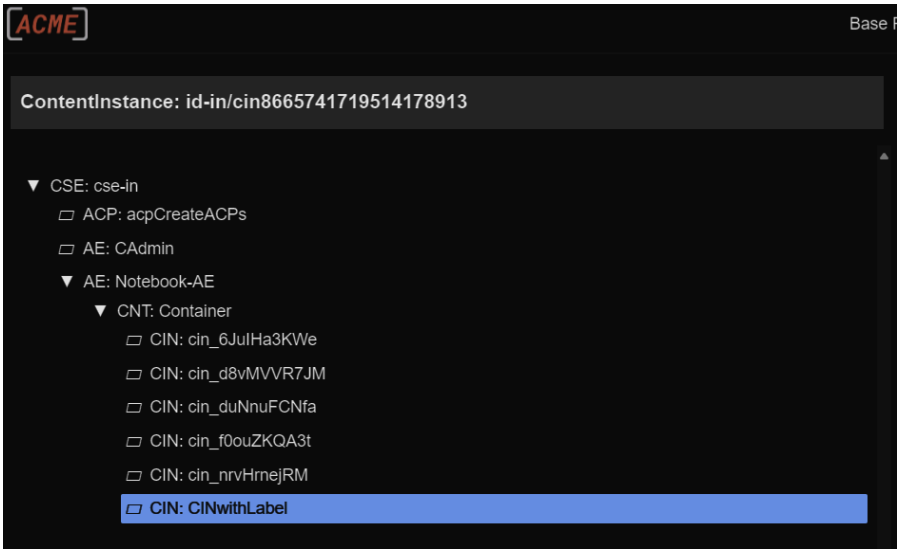

Figure 1:

Noël Jumin
Clément Gauché

Basic Resource Manipulation

Each Jupyter notebook used an initialization script (%run src/init.py) to configure the necessary modules and prepare the CSE for the respective operations. This ensured consistency between resource manipulations.

During these notebooks we discovered how to create and manage basic resources.
The <CSEBase> is the root of the oneM2M resource tree, and the first operation was to send a REST request to retrieve it.

```
RETRIEVE(to=cseBaseName, originator=defaultOriginator)
```

The second operation was to create an <AE> resource, which represents applications or services in the oneM2M architecture.

```
def create_ae():
    url = 'http://localhost:8080/cse-in'
    payload = {
        "m2m:ae": {
            "rn": "Notebook-AE",
            "api": "N-AE",
            "rr": True
        }
    }
    headers = {'X-M2M-Origin': 'CAdmin', 'Content-Type':
'application/json;ty=2'}
    r = requests.post(url, json=payload, headers=headers)
    print(r.status_code, r.json())
```

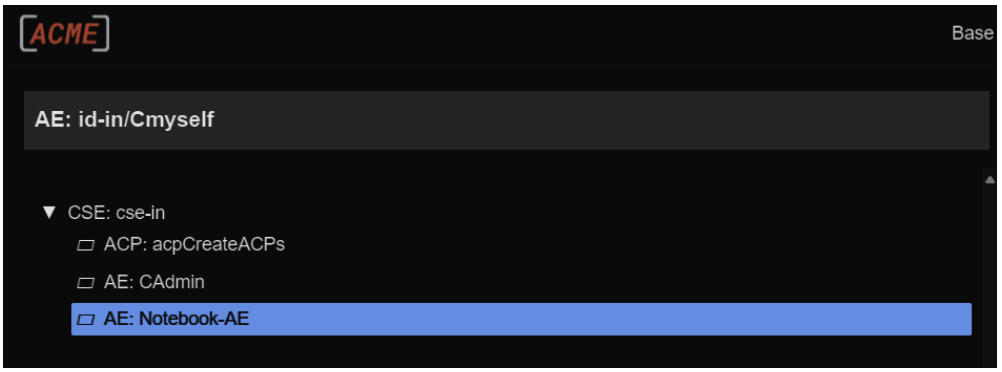

Figure 2: Snapshot of the AE resource created.

The third and last basic operation was the creation and management of <Container> and <ContentInstance> resources.
The <Container> acts as a logical grouping for data instances (<ContentInstance>), which are representing the data payloads.

```
def create_cin():
    url = 'http://localhost:8080/cse-in/Notebook-AE/Container'
```

```
payload = {
    "m2m:cin": {
        "cnf": "text/plain:0",
        "con": "Hello World!"
    }
}
headers = {
    'X-M2M-Origin': 'Cmyself',
    "X-M2M-RI": '123',
    'X-M2M-RVI": "'3',
    "Content-Type': 'application/json; ty-4', # contentInstance
}
r = requests.post(url, json=payload, headers=headers)
print(r.status_code, r.json())
```

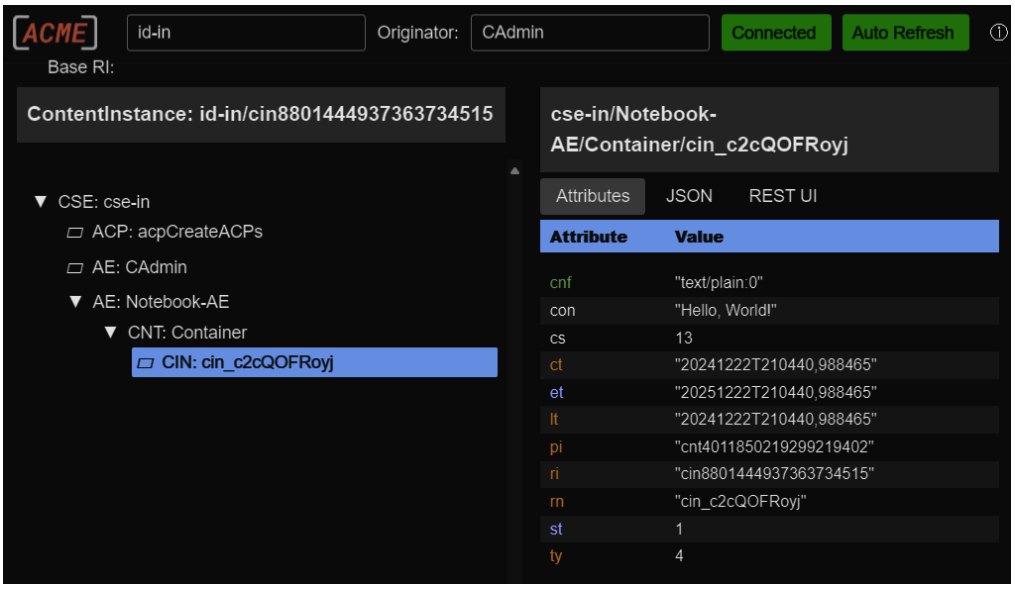

Figure 3: Snapshot of the Container, and ContentInstance resource created.

Noël Jumin
Clément Gauché

Advanced Resource Management

After discovering the basic resources manipulation, the notebooks make us discover more complex actions such as the resource discovery, the <Group> or the <AccessControlPolicy> (ACP) resources.

Resource discovery

Resource discovery in oneM2M allows for efficient querying and filtering of resources based on specific criteria. This functionality is particularly useful when working with large sets of data or when specific resource characteristics need to be identified dynamically. For example, a discovery request can filter resources by their type, such as retrieving only <ContentInstance> resources under a particular <Container>. It can return the result either as references to the resources or the resources themselves, depending on the parameters specified. This flexibility allows discovery to be both powerful and efficient.

```
RETRIEVE(
    to=cseBaseName + '/Notebook-AE/Container',
    filterCriteria={'resourceType': Type.ContentInstance},
    resultContent=RCN.childResources
)
```

<Group> resources

One of the most powerful feature of oneM2M is the <Group> resource, which allows multiple resources to be combined into a single, addressable entity. This is useful when managing collections of resources that need to be controlled or queried together. For example, a <Group> resource could represent all the streetlights in a city block, where each member is a <Container> resource for an individual light. A fan-out point will be automatically created for that group, which may have operations like retrieve or update applied to it such that all the group members will be affected together. This makes large-scale resource management in IoT systems easier.

```
CREATE(
    to=cseBaseName,
    primitiveContent={
        "m2m:Group": {
            "resourceName": "Lights-Group",
            "memberIDs": [
                cseBaseName + '/AE1/Container1',
                cseBaseName + '/AE2/Container2'
            ]
        }
```

```
    }
)
```

<Access Control Policies> resources

Access control in oneM2M is managed by the <AccessControlPolicy> (ACP) resources, which define permissions to access other resources. A policy identifies those originators permitted to perform CREATE, RETRIEVE, UPDATE, or DELETE operations on the protected resources.
By default, full access is granted to the creator of a resource, but this can be overridden by assigning an ACP. For instance, an ACP can provide limited access to a certain user or application while the original creator still retains full access. This feature enhances security and ensures proper governance over shared resources.

```
CREATE(
    to=cseBaseName + '/AE2',
    primitiveContent={
        "m2m:AccessControlPolicy": {
            "resourceName": "ACP-Test",
            "privileges": {
                "accessControlRule": [
                    {"accessControlOriginators": ["CUser1"],
"accessControlOperations": 1}
                ]
            }
        }
    }
)
```

Notifications

The subscription and notification mechanism in oneM2M enables the monitoring of resource changes in an asynchronous manner. Every time a specified event happens, such as the creation, update, or deletion of a child resource, the creation of a <Subscription> resource will notify the user. Notifications can be sent to an URI, enabling external systems to react in real-time. As such, events can be created on changes to a sensor value indicating that an application can process new data.

```
CREATE(
    to=cseBaseName + '/AE2/Container',
    primitiveContent={
        "m2m:Subscription": {
```

```
            "resourceName": "Subscription",
            "notificationURI": ["http://localhost:9999"],
            "eventNotificationCriteria": {"notificationEventType": [1,
2]}
        }
    }
)
```

Specialized Containers

The <flexContainer> resource extends the capabilities of the standard <Container> to support structured data in one resource. This can be especially helpful for an application that requires predefined structure in their data, like RGB color values for a lighting system. Specialized containers, such as [colour], enable all the related data points (for example, red, green and blue) to be grouped and updated at the same time. Besides this, features like versioning can be activated in a way that, for one <flexContainer>, previous states are also kept as <flexContainerInstance> resources. Thus, the resource becomes more capable, ensuring that historical data would be kept accessible and, consequently, manageable.

```
CREATE(
    to=cseBaseName + '/Notebook-AE',
    primitiveContent={
        "cod:colour": {
            "resourceName": "Colour",
            "containerDefinition": "org.onem2m.colour",
            "red": 255, "green": 255, "blue": 0
        }
    }
)
```

Noël Jumin
Clément Gauché

Creation of our application

For our application, we decided to simulate smart home features where notifications are sent when the state of a bedroom lamp is changed. To do that we used the oneM2M framework's subscription and notification mechanism, through two ESP modules acting as IoT devices. One is controlling a button and the other is managing a LED.

Notification Server Setup

In order to enable the notifications, we configured a notification server able to receive updates from ACME.
We can run it by using the command `%run ./NotificationServer.py`
Once launched the server is listening on port 9999 for incoming notification. Upon a change in the lamp's state, the following notification is received:

```
Notification server started.
Listening for http(s) connections on port 9999.
### Received Notification (http)
Host: localhost:9999
User-Agent: ACME 0.10.2
Accept-Encoding: gzip, deflate, br, zstd
Accept: * /*
Connection: keep-alive
Date: Mon, 09 Dec 2024 14:02:32 GMT
Content-Type: application/json
cache-control: no-cache
X-M2M-Origin: /id-in
X-M2M-RI: 3705952827206799903
X-M2M-RVI: 4
X-M2M-OT: 20241209T140232,773822
Content-Length: 82
```

Example 1: Notification of a Button State Change

```
### Received Notification (http)
Host: localhost:9999
User-Agent: ACME 0.10.2
Content-Type: application/json
{
    "m2m:sgn": {
        "new": {
            "rep": {
                "m2m:cin": {
                    "rn": "CIN-ButtonState1",
                    "lbl": ["tag:greeting"],
```

```
                    "cnf": "text/plain:0",
                    "con": "button-on",
                    "ri": "cin123456789",
                    "pi": "cnt987654321",
                    "ct": "20241209T140502,690930",
                    "lt": "20241209T140502,690930",
                    "ty": 4,
                    "cs": 9,
                    "st": 1
                }
            }
        },
        "net": 3,
        "sur": "/id-in/sub49203372518287601 52"
    }
}
```

Example 2: Notification of a LED State Change

```
### Received Notification (http)
Host: localhost:9999
User-Agent: ACME 0.10.2
Content-Type: application/json
{
    "m2m:sgn": {
        "new": {
            "rep": {
                "m2m:cin": {
                    "rn": "CIN-LedState1",
                    "lbl": ["tag:light"],
                    "cnf": "text/plain:0",
                    "con": "light-on",
                    "ri": "cin654321123",
                    "pi": "cnt321654987",
                    "ct": "20241209T141002,783210",
                    "lt": "20241209T141002,783210",
                    "ty": 4,
                    "cs": 8,
                    "st": 2
                }
            }
        },
        "net": 3,
        "sur": "/id-in/sub49203372518287601 52"
    }
}
```

Each notification contains details about the updated state (e.g., button-on or light-on) (“con”), the unique identifiers for the ContentInstance and its parent container (“ri” and “pi”), and the event type (“net”).

Noël Jumin
Clément Gauché

System Setup and Initialization

The system consists of two AE (Application Entities), ESP1, the LED Module which manages the LED state (light-on or light-off), and ESP2, the Button Module, which manages the button state (button-on or button-off).

The setup involves the creation of AEs for the button and the LED, the creation of Containers (ESP1 and ESP2) to store their respective states, and the creation of the Initial Content Instances (CINs) for each container.

The setup is automated through a function setup_environment().

```
def setup_environment():
    createAE("Cmyself_ESP1", "ESP1", 'http://localhost:8080/cse-in')
    createAE("Cmyself_ESP2", "ESP2", 'http://localhost:8080/cse-in')
    createCONT("Cmyself_ESP1", "Led", 'http://localhost:8080/cse-in/ESP1')
    createCONT("Cmyself_ESP2", "Button", 'http://localhost:8080/cse-in/ESP2')
    createCIN("Cmyself_ESP1", "Init", "light-off",
'http://localhost:8080/cse-in/ESP1/Led')
    createCIN("Cmyself_ESP2", "Init", "button-off",
'http://localhost:8080/cse-in/ESP2/Button')
```

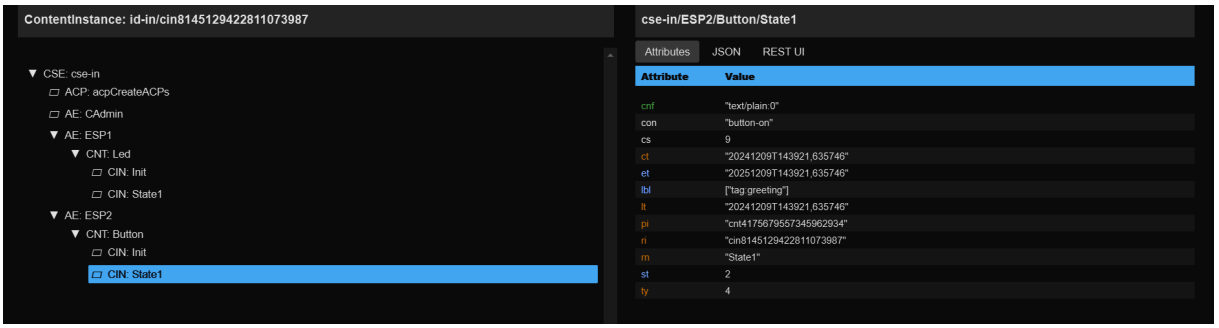

Figure 4: Snapshot of the set up environment

Principal functionalities

The principal functionalities are the button state simulation, the led state update and their parallel execution.The button module (ESP2) simulates periodic changes using the change_state_button() function:

```
def change_state_button():
    i = 0
    while i < 10:
        state = "button-on" if i % 2 else "button-off"
        createCIN("Cmyself_ESP2", f"State{i}", state,
'http://localhost:8080/cse-in/ESP2/Button')
        i += 1
        time.sleep(2)
    stop_threads = True
```

Every two seconds, the button is toggling between button-on and button-off, sending a <ContentInstance> to the ESP2 container:

```
### Received Notification (http)
Host: localhost:9999
User-Agent: ACME 0.10.2
Accept-Encoding: gzip, deflate, br, zstd
Accept: "/"
Connection: keep-alive
Date: Mon, 09 Dec 2024 14:05:02 GMT
Content-Type: application/json
cache-control; no-cache
X-M2M-Origin: /id-in
X-M2M-RI: 2539975737011965370
X-M2M-RVI: 4
X-M2M-OT: 20241209T140502,690930
Content-Length: 374
```

For the led state update, the led module (ESP1) is retrieving the button's latest state and updates its own state accordingly. This is handled by out look_state_button() function:

```
def look_state_button():
    while not stop_threads:
        state_button = retrieve("Cmyself_ESP2",
'http://localhost:8080/cse-in/ESP2/Button/la')
        state_led = "light-on" if state_button == "button-on" else "light-off"
        createCIN("Cmyself_ESP1", f"State{i}", state_led,
'http://localhost:8080/cse-in/ESP1/Led')
```

The use of the <la> virtual resource, allows the function to continuously monitor the button state and updates the LED state (ESP1 container).

And finally to allow the simultaneous execution of the button state simulation, without launching different terminals, we launch them on different threads:

```
thread1 = threading.Thread(target=look_state_button)
thread2 = threading.Thread(target=change_state_button)
thread1.start()
thread2.start()
thread1.join()
thread2.join()
```

Advanced Features

We extended the application functionalities by including optional features such as the **notification subscription** and the **resource grouping**.

We put in place a subscription to the button's container, to allow the button to send notifications to the server whenever the button's state is changed. This removes the need for the LED module to continuously poll the state of the button. We implemented it using the subscribe() function:

```
subscribe(
    "Cmyself_ESP1", "ButtonSubscription",
    'http://localhost:8080/cse-in/ESP2/Button',
    ["http://localhost:9999"],
    NotificationContentType.ALL_RESOURCE.value,
    [NotificationEventType.UPDATE_OF_RESOURCE.value]
)
```

We grouped the ressources into a single group “resource”. It allows for centralized control of all group members.
In our case it doesn’t help much because of the little number of resources to manage.
Due to time we haven’t been able to implement everything we wanted to:

- Setting up a server would be the best solution, since the led change would be called only when a change occurs. Notion of change notification.
- Instead of relying on the state of the button (on or off), we could save the state of the led: a pulse switch instead of a toggle switch, with a view to having several source switches controlling the same led.

Noël Jumin
Clément Gauché

Conclusion

This project provided a comprehensive exploration of the oneM2M framework and its practical applications in IoT systems. Using the ACME CSE, we implemented key features such as notifications, resource grouping and asynchronous communication. This emulates a running smart home system where a button controls a lamp in real time. This project showed how well the oneM2M resource model works. An application entity together with its containers and content instances handled states efficiently.

Notifications enabled the implementation of real-time interactions between button and LED modules without the constant polling mechanism. This approach highlighted the event-driven nature of oneM2M in optimizing resource usage in dynamic IoT environments. The use of modular Python scripts ensured scalability for the system, which can be easily expanded in the future to add more devices or functionalities.

The project has achieved the main goals of the work, but there is always room for improvement. The notification mechanism works well, but it could be further improved by having a dedicated event-handling server for better responsiveness and fewer unnecessary operations. Also, changing the toggle mechanism of the button to a pulse-based control system would make the implementation much more realistic because then several buttons could drive the same LED. This modification would bring the system in line with real-world IoT use cases, such as shared controls in a smart home or office environment.