FAS Assignment 1
DingNet exemplar HTTP interface

Fundamentals of Adaptive Software 2023

Group: 4-1
Member names: [redacted]
Emails: [redacted]
VUnetIDs: [redacted]
Master program and Track: [redacted]

November 12, 2023

Repositories

The code is available at the repositories below (both in the branch `main`).

- **DingNet -** [redacted]
  This is our fork of the DingNet exeplar where we have implemented a HTTP interface to monitor and perform adaptations.

- **UPISAS -** [redacted]
  This is our fork of UPISAS that implements the required changes to get our implementation of the above exemplar to work.

Contents

1 Introduction

For this assignment, we chose the DingNet[1] exemplar, which serves as a reference implementation for research in self-adaptation within the IoT domain, particularly relevant to smart city applications. It includes a simulator that reflects an actual IoT setup in Leuven, Belgium, comprising a network of gateways connected to a user-facing front-end server. These gateways interact with both stationary and mobile motes through a LoRaWAN network. The motes are equipped with sensors and actuators studying and testing various self-adaptive solutions. Our engagement with DingNet focuses on enhancing its functionality by integrating an HTTP interface, thereby deepening our understanding of self-adaptive systems in practical IoT environments.

This report chronologically describes our experiences and implementation of code during assignment 1 of the 2023 course *Fundamentals of Adaptive Software*. It is structured to sequentially outline the key phases of our project. Beginning with the initial setup of the HTTP server, it progresses through the steps involved in running the simulation. We then describe the implementation of various endpoints, such as `/monitor`, `/adaptation_options`, and `/execute`, each discussed in their dedicated sections. Following this, we describe our approach to creating the JSON schema for these endpoints. The report concludes with a section on the finishing touches added to our project. Lastly, we include a division of work section, detailing the individual contributions of our team members.

2 Progress report

The following sections describe in detail our progress during the implementation of the `HTTP` server in the DingNet exemplar, as well as the required changes to UPISAS, in chronological order.

2.1 Basic setup

In the first week, the first task at hand was getting the exemplar and UPISAS running as well as setting up the respective repositories on GitHub. Besides some confusion about the instructions about forking repositories (which would make our repositories public) and dockerization, this part of the setup went relatively smoothly.

Additionally, we made the initial changes to UPISAS to include DingNet in the tests and exemplars. This includes adding "DingNet.py" under "exemplars" and a new directory under "tests" called "dingnet" that has a file "test_dingnet_interface.py" that implements the test cases for the DingNet exemplar.

2.2 Setting up the HTTP Server

After setting up the project and the repositories, we started working on the backbones of the HTTP server. We quickly discovered that Java has a httpserver package we can use. The package is very straightforward to use, and we quickly got the HTTP server up and running. We created classes implementing the httpserver.HttpHandler interface for each of the endpoints and registered them to the server. Extending HttpHandler provides us with the handle method, which will get called once the HTTP server receives a request on the corresponding endpoint. The handlers are in the "src/HTTP" directory. Their filenames and which endpoint they correspond to are shown in Table 1.

After implementing the base of the server, we wanted to test if the connection between UPISAS and our endpoints could be established. To test this, we built a jar artifact of the HTTP server and dockerized it into an image using a Dockerfile. The Dockerfile uses "openjdk:11" as a base, copies the jar file into the image, and describes the command that should be executed: `java -jar DingNet.jar`. After making the required changes to the UPISAS framework, such that it would use the docker image we just created and expose the right ports, we noticed that the first test UPISAS executes is a request to "/" to ensure a connection can be made. Table 1 contains BaseMapping for this specific reason. After implementing this, we verified that a connection between UPISAS and our DingNet HTTP interface was successfully established.

Filename	Endpoint
BaseHandler.java	/
MonitorHandler.java	/monitor
MonitorSchemaHandler.java	/monitor_schema
AdaptationOptionsHandler.java	/adaptation_options
AdaptationOptionsSchemaHandler.java	/adaptation_options_schema
ExecuteHandler.java	/execute
ExecuteSchemaHandler.java	/execute_schema
StartRunHandler.java	/start_run

Table 1: Endpoints and the corresponding filename they are implemented in. Each file contains a class of the same name as the file that implements the endpoint.

2.3 Running the simulation

Next up, we had to run the simulation such that we could retrieve data from it while it was running for the monitor endpoint and modify data with the execute endpoint. We found the `MapView` class, which was a barebones simulation run that showed the map with the motes and gateways on a GPS map and showed some graphs when the simulation run finished with statistics from the motes' sensors over time. We decided to use this class to do our simulation run. In order to do so, we removed all the GUI-related components since those would not work inside a docker container and are not required for the functionality of the endpoints. Additionally, we renamed it to `MainSimulation`, which is available under the "Simulation" directory.

Now that we had a simulation to run, we were faced with the problem of running the simulation concurrently with the `HTTP` server. There were two options for this:

1. Starting the simulation as a separate *process.* To share data between the `HTTP` server and the simulation, we would then have to use a database, communicate with signals, or use a socket.

2. Starting the simulation as a separate *thread.* Starting the simulation as a thread reduces the complications with communicating between the `HTTP` server and the simulation because they would then live in the same address space. However, it does require some sort of synchronization mechanism to ensure there are no race conditions.

In the end, we decided to use a thread for the simulation because implementing thread synchronization in Java is fairly trivial (especially if performance is not the primary concern) by using the `synchronized` keyword or by using `AtomicReference`s, and implementing any of the synchronization schemes necessary for a separate process would be far more time-intensive.

Another thing to consider was when and how to start the simulation. There were again two options:

1. Start the simulation when the `HTTP` server starts. With this approach, the server needs to be restarted every time we want to do a run. This is the behavior of the UPISAS tests, but it may be undesirable for testing purposes and ease of use.

2. Start the simulation at a later point. To do so, we could, for example, add another endpoint to the `HTTP` server that starts the simulation.

We decided to go with option 2 and added the `start_run` endpoint that creates a new thread in which it starts the simulation. The next step was to find a way to communicate between the running simulation and the `HTTP` server, such that we could retrieve part of the execution state for the `monitor` endpoint and such that we could modify the values in the `execute` endpoint. To do so, we created a new class called `SimulationState`, a reference to an instance of which is then shared between the `HTTP` server (and all its handlers) and the simulation. As mentioned earlier, to prevent the race conditions due to concurrency between the two threads, there are two options: we either use `AtomicReferences` or the `synchronized` keyword. An `AtomicReference` wraps a reference to an object to ensure that all *reads* and *writes* to this reference are atomic operations, meaning no *reads* and *writes* can occur at the same time, preventing race conditions. The `synchronized` keyword surrounds a code block and creates a lock around the code it wraps. By placing these blocks around the pieces of code accessed by multiple threads, we can prevent race conditions. In the end, we went with the `synchronized` approach because with a library called "Lombok" it was very straightforward to add the synchronization on all getters and setters in the `SimulationState`.

2.4 Implementing the endpoints

2.4.1 HTTPResponse Class

We use the `HTTPRespones` class to send the responses for the `HTTP` requests. The class acts as a container and consists of an `HTTP` status code and a body. These are then used to write a response back to the client. This class was added as a convenience class to make the `Handler` classes more readable.

2.4.2 /monitor

The `monitor` command is implemented in the class `MonitorHandler`. As mentioned earlier, we introduced a class called `SimulationState`. This class is used to store the environment of DingNet, a boolean variable to indicate whether the simulation is running or not, and two classes `GatewayState` and `MoteState`. These last two classes store information about values that are monitorable during the whole simulation. The values of the motes and gateways that can be monitored, as defined in `GatewayState` and `MoteState`, can be found in Table 2 and Table 3. These fields were derived from the DingNet framework and we decided to include the fields that are needed to create an instance of the `Mote` and `Gateway` classes. The `SimulationState` class is initialized once the simulation starts.

Since `SimulationState` contains the current state of the simulation execution, we can directly use it in our monitor endpoint. When the `monitor` command is executed, `SimulationState` is turned into a JSON object and included in the body of the `HTTP` response. This is then sent back to the client along with the `HTTP` status code `200 OK`.

Name	Description
`EUI`	The gateway identifier.
`XPos`	The x-coordinate of the gateway.
`YPos`	The y-coordinate of the gateway.
`SF`	The spreading factor of the gateway.
`Transmission Power`	The transmission power of the gateway.

Table 2: The values of the gateways that are monitored.

Name	Description
`id`	The unique id of the mote used for the simulation.
`EUI`	The device's unique identifier.
`XPos`	The x-coordinate of the mote.
`YPos`	The y-coordinate of the mote.
`SF`	The spreading factor of the mote.
`Transmission Power`	The transmission power of the mote.
`Sensors`	A list of sensors on the mote.
`Energy Level`	The energy level of the mote.
`Path`	The path the mote will follow.
`Sampling Rate`	The sampling rate of the mote.
`Movement Speed`	The movement speed of the mote.
`Start Offset`	The start offset of the mote.
`Shortest Distance To Gateway`	The distance to the nearest gateway.
`Highest Received Signal Reference`	The highest received signal by any of the gateways for a given mote.

Table 3: The values of the motes that are monitored.

2.4.3 /adaptation_options

`HTTP` requests to the endpoint `adaptation_options` are handled by the class `AdaptationOptions-Handler`. To define the adaptation options, we introduced a new model class `AdaptationOptionModel` with four members describing the name, a description, the minimum value, and the maximum value of the adaptation option. The specific five `AdaptationOptionModel` instances with the names `power`, `spreading_factor`, `sampling_rate`, `movement_speed`, and `energy_level` were statically defined within the `AdaptationOptionsHandler`. When receiving an `HTTP` request, the object mapper converts the name and value of each member of the `AdaptationOption` instances to a key-value pair in a *JSON* string. That string, as shown in Listing 1, is returned in the body of the `HTTP` response. All adaptation options are listed in an array under the key `"items"`. The descriptions, minimum and maximum values were mostly retrieved from the Dingnet paper [1]. Some minimum values were also determined from the context of the parameter. If no maximum value was specified in the paper, the `"maxValue"` key is not included in the output. Note that due to the internal implementation of the parameter values as Integers in JAVA, there is a factual upper limit to the values that can be assigned.

The handler always responds with the `HTTP` status code `200 OK`.

```json
{
    "items": [
        {
            "name": "power",
            "description": "Determines the energy consumed by the mote for communication.",
            "minValue": -1.0,
            "maxValue": 15.0
        },
        {
            "name": "spreading_factor",
            "description": "Determines the duration of the communication",
            "minValue": 0.0,
            "maxValue": 12.0
        },
        {
            "name": "sampling_rate",
            "description": "The sampling rate of the sensors of the mote.",
            "minValue": 0.0
        },
        {
            "name": "movement_speed",
            "description": "The movement speed of the mote.",
            "minValue": 0.0
        },
        {
            "name": "energy_level",
            "description": "The energy level of the mote.",
            "minValue": 0.0
        }
    ]
}
```

Listing 1: The body of the `HTTP` response of the `adaptation_options` endpoint.

2.4.4 /execute

The `ExecuteHandler` class was created to handle the requests to the `execute` endpoint. After using a different JSON structure for the `HTTP` request, we decided to align the JSON structure with the response of the `adaptation_options` endpoint. Specifically, the request body has to contain a JSON object with the key `"items"`. The value associated with that key is an array with a separate object for each mote to which adaptions are to be applied. In each of those objects, the mote is specified by an Integer value associated with the key `"id"`. The adaptation options for a mote are specified by an array after the key `"adaptations"`. As specified in Section 2.4.3, each object of that array must have the keys `"name"` and `"value"`. An example of this request body can be found in Listing 2 below.

```json
{
    "items": [
        {
            "id": 0,
            "adaptations": [
                {
                    "name": "power",
                    "value": 10
                },
                {
                    "name": "sampling_rate",
                    "value": 100
                }
            ]
        },
        {
            "id": 1,
            "adaptations": [
                {
                    "name": "power",
                    "value": 5
                },
                {
                    "name": "spreading_factor",
                    "value": 10
                }
            ]
        }
    ]
}
```

Listing 2: An example request body for the `execute` endpoint.

The ExecuteHandler validates the provided mote ID ensuring it falls within the valid range of the existing mote IDs in the simulation. In addition to checking for valid mote IDs, the handler also validates adaptation names and their respective values against predefined ranges. If an illegal parameter is detected, such as an invalid mote ID, an unrecognized adaptation name, or an adaptation value that is out of the permissible range, the handler responds with the HTTP status 400 BAD REQUEST. The nature of the error is indicated in the response body, for example, "Malformed input.", "Invalid mote ID.", "Invalid adaptation name.", or "Adaptation value out of range.").

In the first version, the handler wrote the incoming adaptations to an instance of AdaptationState. That class was introduced to represent the current state of all adaptations. The adaptations were then applied in the main loop of the simulation. Since we did not want the mote effector to apply the adaptions in every iteration of the loop, a Boolean flag changed was used to indicate if the value of the adaptations had recently been changed. Later, we moved the application of the adaptions in the ExecuteHandler object, thus obviating the need for the AdaptationState class. To realize that, we included the environment in the shared SimulationState instance so the ExecuteHandler could access the motes of the simulation. Upon receiving a valid request, the handler iterates over each mote specified in the input JSON. For each mote, it retrieves the corresponding list of adaptations and applies them using the MoteEffector class. The MoteEffector is responsible for implementing the actual changes on the motes, such as setting power levels, sampling rates, spreading factors, movement speeds, or energy levels.

After successfully applying all specified adaptations, the handler constructs a success response, indicating that the adaptations have been applied. This response is returned to the client with an 200 OK HTTP status.

2.5 Creating the JSON schema endpoints

Now that we had implemented the functionality of all the base endpoints, we also had to implement the JSON schema endpoints. We could have done this by hand. However, in favor of extensibility and ease of use, we decided to use a library to generate the JSON schemas on the fly. The library is called jsonschema-generator, and it provides methods to create a JSON schema from any Java class. We created a method called toJsonSchema in a class called Schema that uses the jsonschema-generator to convert a class to a JSON schema. We use the latest JSON schema draft version: 2020-12 [1].

As discussed in Section 2.4.2, the monitor endpoint returns the SimultationState. Therefore, the implementation of the simulation_schema endpoint consists of converting this class to the JSON schema using the aforementioned library.

Similarly, for the adaptation_options_schema and execute_schema endpoints, we converted a list containing AdaptationOptionsModels and a list containing ExecuteModels respectively. However, when testing with UPISAS, we noticed that it expects an object, not an array, in the JSON schema validation. Therefore, we introduced two new classes: AdaptationOptionsDTO and ExecuteDTO, which each have a items field holding the list of models. This introduced some small changes in the endpoint implementation that we skipped over in Section 2.4. All the aforementioned models can be found in the models directory.

[1] https://json-schema.org/draft/2020-12/schema

2.6 Finising touches

Lastly, we made some changes to UPISAS to get our tests fully working. This includes a loop in the `start_run` function that waits for the `HTTP` server to start by waiting until the start message is printed in the docker container's log.

Additionally, we had to adjust the input for the `execute` interface in the `test_dingnet_interface` file, because it was by default using input for a different exemplar and therefore getting wrong results.

With these changes, all the tests are now passing and working as intended.

2.7 Libaries

Table 4 contains a list of libraries we used, the names of the library files it uses, and a short description on why we included it and what it is for. The libraries can be found in the "lib" folder as ".jar" files. Note that this list only includes libraries we added on top of the existing ones from the exemplar.

Name	Required libraries	Description
Jackson	`jackson-core`, `jackson-databind`, `jackson-annotations`[2]	Jackson provides us with the `ObjectMapper` class, which allows us to write Java objects into JSON and vice versa. We use this in many of the endpoints. For example, we return the `SimulationState` as JSON in the `monitor` endpoint. In `execute`, we do the reverse operation: we read the JSON provided and read it into a Java class. The `jackson-annotation` package allows us to annotate fields and classes with annotations to change the behavior of the object mapper. For example, we added `@JsonIgnore` to the `Environment` in `SimulationState` such that it is not returned in the `monitor` endpoint.
JSON Schema Generator	`jsonschema-generator`, `jsonschema-module-jackson`, `slf4j-api`, `slf4j-nop`, `classmate`	As described in Section 2.5, we use this library to convert our Java classes into a JSON schema representation. It has SLF4J and an SLF4J implementation (we use `slf4j-nop`) as dependencies for logging and `classmate` for the class inspecting.
Project Lombok	`lombok`[3]	Lombok provides a set of convenience annotations that reduce the amount of boilerplate code required in Java classes. We mostly used it to create constructors, getters, and setters, but also to add the `@Synchronized` annotation on all the getters and setters of the `SimulationState` class. This way, all the getters and setters are wrapped with the `synchronized` keyword, which (as explained in Section 2.3) is required for concurrency.

Table 4: The libraries we used for implementing the `HTTP` server and endpoints.

[2] Annotation processing must be explicitly enabled in some IDE's to handle these annotations.
[3] In addition to the jar file, lombok requires the lombok plugin in order to work properly in most IDE's.

3 Division of work

Functionality	Team member(s)
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]

References

[1] M. Provoost and D. Weyns, "Dingnet: A self-adaptive internet-of-things exemplar," in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2019, pp. 195–201.