



CrowdNav

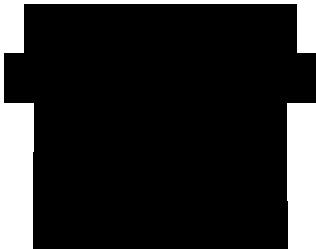


Fundamentals of Adaptive Software

Name of the Assignment

Assignment 1: Exemplar Interface

Report

Project group 7.1

Name	Student Nr	Email
		

Date:
Amsterdam, Friday 10 November, 2023

Contents

1	Introduction	3
2	Implementation of the HTTP Interface	3
2.1	Monitor	5
2.1.1	Monitor Endpoint	5
2.1.2	Monitor Schema Endpoint	6
2.2	Adaptation	8
2.2.1	Adaptation Options Endpoint	8
2.2.2	Adaptation Options Schema Endpoint	9
2.3	Execute	11
2.3.1	Execute Endpoint	12
2.3.2	Execute Schema Endpoint	12
3	UPISAS Testing	13
4	Challenges and Solutions	14
5	Conclusion	15
6	Division of Work	16

1 Introduction

The following report serves as explanatory support for the Assignment 1 project, Fundamentals of Adaptive Software at Virje University of Amsterdam. The project is built on top of the existing adaptive software solution CrowdNav. CrowdNav simulates a system consisting of a number of cars traveling in a city following the itineraries of their drivers and of a centralized navigation service. The main goal of self-adaptation in CrowdNav relates to the non-functional requirement of optimizing user satisfaction, the adaptation happens around trip duration and user complaint rate. [1] Additionally, we also have used the *managing system* UPISAS, which at this stage provided a series of tests to check the developed implementation.

This report aims to provide an in-depth understanding of the http interface that allows users to interact with the CrowdNav simulation through six endpoints. These endpoints are: **monitor** - which allows the user to monitor the values, **adaptation options** which allows the user to see the options for simulation adaptation, and **execute** which allows the user to change the simulation values. Each endpoint has a respective JSON schema endpoint, that informs the user about the structure of GET or PUT .

Our work is delivered in the form of two private GitHub repositories. One is the CrowdNav simulation including a server and the latter is the forked UPISAS project with tests for our implementation. Both GitHub repositories include README files and can be run through Docker to simplify the assessment^{1 2}.

The report is structured as follows. First, we give the details about HTTP server implementation, then we focus on the UPISAS testing that is provided in a separate repository. After that, we elaborated on the challenges and solutions that we faced during the project. Finally, we provide a conclusion and decision of the work section. Within the report, we provide multiple design decisions and rationale for them in the form of a fixed table in order to enable easier understanding for a reader.

2 Implementation of the HTTP Interface

In this section, we want to explain the HTTP Server and endpoints we developed for implementing the communication between Upisas and CrowdNav. In order to capture data from the simulation in CrowdNav and send it to our HTTP endpoints, we decided to create a singleton class called SimulationData containing all the traffic data required from the simulation. The rationale behind using a singleton class was having one single instantiation within the entire project. This enabled us to keep the data consistent and always pass the most updated values to the HTTP endpoints. SimulationData contains all the required data as fields as well as the setter and getter methods for changing and reading values. This design decision is also described in the table below.



Design Decision

Use of a Singleton class for Data Transfer

Rationale

Centralized Data Management: The choice of a singleton class consolidates all necessary traffic data in one place. This facilitates easy access to this data from any part of the project, ensuring a centralized and consistent source for information exchange. Having a single instance of `SimulationData` means that all components communicate with and retrieve data from the same source, reducing the risk of discrepancies due to multiple instances. The setter and getter functions provided within the singleton class enable controlled modification and retrieval of data, maintaining encapsulation and controlled access to the information.

Consistency and Synchronization: By ensuring a single instance of `SimulationData` (Singleton design), it guarantees data consistency. With multiple instances, it becomes challenging to maintain synchronized and updated information across the project.

Scalability and Extensibility: The singleton class design promotes scalability, making it easier to expand or modify the traffic data structure if needed. As new requirements arise or as the project evolves, having a centralized data management approach can be more adaptable and extendable, as changes can be implemented within a singular structure.

However, having this singleton was not sufficient for HTTP data streaming between two different projects as they are run as two processes. Therefore, we decided to add a JSON file as a middleware for storing the data. Within the setter functions of the class, the data is always written to this JSON file after any change or update. This way we ensured that data is always consistently stored and can be accessed. This design decision is also described in the table below.

Design Decision

Using a JSON file Consistent Data Storage

Rationale

Inter-Process Communication: The need for data transmission across separate processes necessitated an intermediary medium. Using a JSON file served as a pragmatic solution for communication between the distinct projects, enabling them to exchange and access the required information easily.

Persistent Data Storage: By implementing a JSON file, the data becomes persistently stored and accessible. This ensures that the information persists beyond the runtime of individual processes. Any updates or changes made to the data within the singleton class's setter functions are immediately reflected by writing the altered information to the JSON file.

Data Consistency: Writing data to the JSON file after every modification within the singleton class ensures that the stored information is consistently updated. This design choice guarantees that any component or process requiring access to this shared information can retrieve the most recent version from the JSON file, thereby maintaining data consistency and accuracy across the projects.

The dynamic data that the user would have an interest in monitoring, are set in two Python scripts `Simulation.py` and `Car.py` files. Therefore, we identified the points where the data gets updated, initialize the `Simulation Data` class, and set the concrete data. This resulted in applying the same changes to our `SimulationData` class. This way we would be able to always transfer the most updated data of the simulation HTTP endpoints. The relevant updates are commented on the provided GitHub repositories.

2.1 Monitor

In this section we explain endpoints implemented in our HTTP server for monitoring capabilities. They allow users to observe the values produced by the system.

2.1.1 Monitor Endpoint

The first endpoint in this section is the monitor endpoint (`/monitor - GET`). This endpoint returns a JSON object with a list of values of everything monitorable within the CrowdNav project. This data includes all the fields existing in the `knobs.json` file as well as four new fields taken from `Car.py` and `Simulation.py` files. The data from the `knobs.json` is depicted in Listing 1.

1 {

```

2  "routerRandomSigmaDescription": "the randomization sigma of
   edge weights",
3  "routerRandomSigma": 0.0,
4  "explorationPercentageDescription": "the percentage of
   routes we want to use for exploration",
5  "explorationPercentage": 0.0,
6  "maxSpeedAndLengthFactorDescription": "how much the length/
   speed influences the routing",
7  "maxSpeedAndLengthFactor": 1,
8  "averageEdgeDurationFactorDescription": "how much the
   averageEdgeFactor influences the routing",
9  "averageEdgeDurationFactor": 1,
10 "freshnessUpdateFactorDescription": "how much the
   freshnessUpdateFactor influences the routing",
11 "freshnessUpdateFactor": 10,
12 "freshnessCutoffValueDescription": "if data is older than
   this we do not consider it in the algorithm",
13 "freshnessCutoffValue": 90,
14 "reRouteEveryTicksDescription": "check for a new route very
   x times after the car starts",
15 "reRouteEveryTicks": 60
16 }

```

Listing 1: Konbs.json Data

The additional data taken from Car.py and Simulation.py files is depicted in listing 2.

```

1  # Time taken for car process.
2  routingDuration
3  # The time of the last simulation (last tick)
4  tickDuration
5  # The number of steps in the simulation
6  tickNumber
7  # Actual VS the theoretical trip duration
8  tripOverhead

```

Listing 2: Simulation Generated Data

2.1.2 Monitor Schema Endpoint

The second endpoint is the Monitor schema (**/monitor_schema - GET**). The schema is defined and implemented to enforce constraints, making the validation process more robust and manageable. For this schema, we added a short description to give context about the meaning of the data and its value as well as a type for each field. The monitor schema is illustrated in listing 3.

```
1 {
2   "type": "object",
3   "properties": {
4     "routingDuration": {
5       "description": "Time it takes for routing",
6       "type": "integer"
7     },
8     "tickDuration": {
9       "description": "The time of the last simulation (
last tick).",
10      "type": "integer"
11    },
12    "tickNumber": {
13      "description": "The number of steps in the
simulation.",
14      "type": "integer"
15    },
16    "tripOverhead": {
17      "description": "The actual duration of a trip
versus the theoretical case where routing is performed
based only on static map data length and maximum speed
of each street.",
18      "type": "number"
19    },
20    "routeRandomSigma": {
21      "description": "The randomization sigma of edge
weights.",
22      "type": "number"
23    },
24    "explorationPercentage": {
25      "description": "The percentage of routes we want
to use for exploration.",
26      "type": "number"
27    },
28    "maxSpeedAndLengthFactor": {
29      "description": "How much the length/speed
influences the routing.",
30      "type": "integer"
31    },
32    "averageEdgeDurationFactor": {
33      "description": "How much the averageEdgeFactor
```



```

    influences the routing.",
    "type": "integer"
  },
  "freshnessUpdateFactor": {
    "description": "How much the
freshnessUpdateFactor influences the routing.",
    "type": "integer"
  },
  "freshnessCutOffValue": {
    "description": "If data is older than this we do
not consider it in the algorithm.",
    "type": "number"
  },
  "reRouteEveryTicks": {
    "description": "Check for a new route every x
times after the car starts.",
    "type": "integer"
  }
}

```

Listing 3: Monitor Schema

2.2 Adaptation

In this section, we describe endpoints implemented in our HTTP server for adaptation capabilities.

2.2.1 Adaptation Options Endpoint

The first endpoint in this part is the adaptation options endpoint (**/adaptation_options - GET**). This endpoint returns a JSON object with all the available adaptation options. These are the configurable aspects of the system. In CrowdNav, the main goal of self-adaptation relates to the non-functional requirement of optimizing user satisfaction. Therefore all the configurable data from knobs.json file is returned in this endpoint. For each field, a short description and a range of values is included in the response to define each options more clear. The ranges of values have been determined by the description in [1], and are as follows:

- reRouteEveryTicks - integer value - no range applied
- routeRandomSigma - float value - no range applied
- explorationPercentage - float value - range 0-30

- maxSpeedAndLengthFactor - integer value - range 1-5 (w-static value from [1])
- averageEdgeDurationFactor - integer value - range 1-5 (w-dynamic value from [1])
- freshnessUpdateFactor - integer value - range 10-20 (w-benefit value from [1])
- freshnessCutOffValue - float value - no range applied

2.2.2 Adaptation Options Schema Endpoint

The second endpoint is the adaptation options schema (`/adaptation_options_schema - GET`). The schema is defined and implemented to enforce constraints, making the validation process more robust and manageable. For this schema, we added a description and type of the value. Additionally, maximum and minimum value properties for the field that have a range. The schema was designed in such a way to inform the users of this endpoint what fields can be adapted, what properties each field contains and what are the correct data types for defining these properties. The adaptation option schema is illustrated in Listing 4.

```
1 {
2   "type": "object",
3   "properties": {
4     "routeRandomSigma": {
5       "type": "object",
6       "properties": {
7         "domain": {
8           "type": "string"
9         }
10      },
11     "description": "Sets the randomization sigma of
edge weights."
12   },
13   "explorationPercentage": {
14     "type": "object",
15     "properties": {
16       "minimum": {
17         "type": "number"
18       },
19       "maximum": {
20         "type": "number"
21       },
22       "domain": {
23         "type": "string"
24       }
25     },
26   },
27 }
```

```
26         "description": "Sets the percentage of routes we
27         want to use for exploration."
28     },
29     "maxSpeedAndLengthFactor": {
30         "type": "object",
31         "properties": {
32             "minimum": {
33                 "type": "integer"
34             },
35             "maximum": {
36                 "type": "integer"
37             },
38             "domain": {
39                 "type": "string"
40             }
41         },
42         "description": "Sets how much the length/
43         speed influences the routing."
44     },
45     "averageEdgeDurationFactor": {
46         "type": "object",
47         "properties": {
48             "minimum": {
49                 "type": "integer"
50             },
51             "maximum": {
52                 "type": "integer"
53             },
54             "domain": {
55                 "type": "string"
56             }
57         },
58         "description": "Sets how much the
59         averageEdgeFactor influences the routing."
60     },
61     "freshnessUpdateFactor": {
62         "type": "object",
63         "properties": {
64             "minimum": {
65                 "type": "integer"
66             }
67         }
68     }
```

```
65         "maximum": {
66             "type": "integer"
67         },
68         "domain": {
69             "type": "string"
70         }
71     },
72     "description": "Sets how much the
freshnessUpdateFactor influences the routing."
73 },
74     "freshnessCutOffValue": {
75         "type": "object",
76         "properties": {
77             "domain": {
78                 "type": "string"
79             }
80         },
81         "description": "Sets the value so that if the
data is older than this we do not consider it in the
algorithm."
82     },
83     "reRouteEveryTicks": {
84         "type": "object",
85         "properties": {
86             "domain": {
87                 "type": "string"
88             }
89         },
90         "description": "Sets the value to check for a new
route every x times after the car starts."
91     }
92 }
```

Listing 4: Adaptation Option Schema

2.3 Execute

In this section, we describe endpoints implemented in our HTTP server for execute capabilities.

2.3.1 Execute Endpoint

The first endpoint in this part is the adaptation options endpoint (**/execute - PUT**). This endpoint executes one or multiple adaptation option in the system. A JSON object originating from the adaptation options endpoint is included in the body of this HTTP request, specifying the adaptation that needs to be executed. All the fields specified in the adaptation options endpoint must be passed as the body of this request. If the value of any field needs to change it can be reflected in the body of this request. In other cases, the values remain unchanged. Just like the response of the adaptation options endpoint, the description and value of each field need to be added to the request body. In case of passing a correct request body, the execution takes place successfully and the endpoint returns a 200 OK response.

2.3.2 Execute Schema Endpoint

The second endpoint is the execute schema (**/execute_schema - GET**). The schema is defined and implemented to enforce constraints, making the validation process more robust and manageable. For this schema, we added description and type for each field. The schema was designed in such a way to inform the users of this endpoint about the nature and type of the execution options. The execution schema is illustrated in Listing 5.

```
1 {
2   "type": "object",
3   "properties": {
4     "routeRandomSigma": {
5       "description": "Sets the randomization sigma of
6 edge weights.",
7       "type": "number"
8     },
9     "explorationPercentage": {
10      "description": "Sets the percentage of routes we
11 want to use for exploration.",
12      "type": "number"
13     },
14     "maxSpeedAndLengthFactor": {
15      "description": "Sets how much the length/speed
16 influences the routing.",
17      "type": "integer"
18     },
19     "averageEdgeDurationFactor": {
20      "description": "Sets how much the
21 averageEdgeFactor influences the routing.",
22      "type": "integer"
23     }
24   }
25 }
```

```

20     },
21     "freshnessUpdateFactor": {
22         "description": "Sets how much the
freshnessUpdateFactor influences the routing.",
23         "type": "integer"
24     },
25     "freshnessCutOffValue": {
26         "description": "Sets the value so that if the
data is older than this we do not consider it in the
algorithm.",
27         "type": "number"
28     },
29     "reRouteEveryTicks": {
30         "description": "Sets the value to check for a new
route every x times after the car starts.",
31         "type": "integer"
32     }
33 }
34 }

```

Listing 5: Execute Schema

3 UPISAS Testing

Our changes for UPISAS are in the GitHub repository ³ and for CrowdNav they are in the GitHub repository ⁴. The following are the detailed steps to test the solution.

- Clone our CrowdNav repository:

```
git clone https://github.com/crowdnav/crowdnav
```

- Clone also our UPISAS repository:

```
git clone https://github.com/upisas/upisas
```

- Get Docker running on your OS.
- The next step is to navigate to the parent directory of the cloned CrowdNav project and build the Docker image:
`docker build -t crowd-nav-local .`

```
cd crowdnav
docker-compose up
```

4 CHALLENGES AND SOLUTIONS

- Next, we go to the parent directory of the cloned UPISAS in a terminal and execute:
`pip install -r requirements.txt`
- Still in the parent directory, now let's run our exemplar tests:
`python -m UPISAS.tests.your_exemplar.test_your_exemplar_interface`
- Before running the tests, please make sure there is no container with the name - "upisas-crowdnav" of our image running, if so please delete it.
- Now all the tests run and pass.

Below, we describe the changes made in UPISAS to enable running our CrowdNav Docker image and the tests successfully.

In UPISAS/exemplars/your_exemplar.py, we added our image name and container name as well as called `start_container()` function to start our container if the autostart is False.

Furthermore, in UPISAS/tests/your_exemplar/test_your_exemplar_interface.py, two changes have been made. Firstly a sleep time of 40 seconds `time.sleep(40)` for the tests: `test_monitor_successfully()` and `test_schema_of_monitor()`. This was conducted in order to make sure the tests started only after the simulation in CrowdNav had started. Another way to achieve this could have been by having an HTTP endpoint to poll and check if the simulation has started in CrowdNav before starting the UPISAS tests. However, the decision has been made to implement the former, since it was a simple and straightforward solution that works across multiple os systems.

Secondly, in the tests: `test_execute_successfully()` and `test_schema_of_execute()`, the PUT request body pertaining to our exemplar was added.

4 Challenges and Solutions

The issues that were faced during the project were related to Docker image implementation, SUMO GUI, Python version, and JSON schema definition. In the following section, we describe in detail the issues and maturation we undertook to address them.

Several challenges emerged while building the Docker image for CrowdNav, and these issues were not unique to our group; other teams encountered similar difficulties. In response to these challenges, we adopted a strategy to address the problem by modifying the Dockerfile. Instead of creating an image from scratch, we decided to build upon an existing Dockerfile, which offers several advantages in terms of efficiency and compatibility. We based our image creation on the foundation of the *starofall/crowdnav:latest* Docker image. This approach allowed us to retain the core configuration and dependencies of CrowdNav while making targeted modifications to specific folders. By customizing only the contents of these specific directories, we could tailor the Docker image to our precise requirements for running the CrowdNav application. This method not only simplified the image creation process but

also ensured that we maintained compatibility with the existing CrowdNav environment, streamlining development and deployment efforts.

By building on top of this system, it's worth noting that we didn't have access to the SUMO GUI, as it wasn't included in the Docker image. However, this wasn't a significant challenge for our purposes. We effectively operated the simulation through the command-line interface, using the terminal to monitor and analyze the simulation and its outputs.

Additionally, in our project, we faced a couple of challenges due to the differences between Python 2.7 and Python 3.x. Examples of this are print statements as well as unsupported or deprecated libraries for Python 2.7. To overcome these challenges, we switched our interpreter and configured our IDE to easily switch between Python 2.7 and Python 3.x interpreters as well as using code assistant tools supporting Python 2.7. This allowed us to write our code in Python 2.7 without much hassle.

Moreover, we faced the challenge of not knowing the precise details of the JSON schema. To tackle this, we looked to established standards that use elements like "value" and "description" within the JSON schema as a reference. This approach helped us structure our schema effectively and maintain alignment with industry best practices. The final challenge was for data persistence when writing and reading simulation data continuously. Updating a singleton with simulation values and consistently writing to a JSON file through the singleton (SimulationData) provides a more persistent and reliable data management approach in Python. By channeling all data updates through the singleton, we establish a controlled and synchronized process, ensuring that data integrity is maintained throughout. This method minimizes the risk of data corruption or inconsistencies that might occur when directly accessing a JSON file, especially in concurrent or multi-threaded environments. It offers a clear and centralized data update point, reducing the chances of conflicts and enabling better error handling. This approach not only enhances data persistence but also simplifies debugging and maintenance, making it a more robust and dependable strategy for managing simulation data.

5 Conclusion

Overall, in the above project, we were able to familiarise ourselves with the principles of adaptive systems and extend the existing framework with HTTP server. The project builds onto the existing CrowdNav system a self-adaptation in large-scale software-intensive distributed systems. This report has outlined the implementation of the HTTP interface, for communication with CrowdNav system. The project has developed six endpoints. (monitor, execute, adaptation options and respective schemas). Additionally, we also reported on the tests that were developed thanks to the UPISAS project. We discussed the design decisions and the rationale behind them, as well as we summarised the challenges that were faced in the development process. Both projects will be further developed in the following assignments.

6 Division of Work

In the following section, the different responsibilities are displayed. We want to emphasise that we always worked and checked quality together, which is why the responsibilities have been shared very equally.

Overall idea development	X	X	X	X
Fixing Docker issues to run CrowdNav simulation		X		
Monitor and Monitor schema development			X	X
Adaptation and adaptation schema development	X	X		
Execute and execute schema development		X		X
Report structure	X		X	
Report: Introduction, Conclusion and Group work	X			
Report: Implementation of the HTTP Interface				X
Report: UPISAS Testing		X		
Report: Challenges and Solutions			X	
UPISAS Testing		X	X	
Github repositories preparation	X			X

References

- [1] Sanny Schmid, Ilias Gerostathopoulos, Christian Prehofer **and** Tomas Bures. “Self-Adaptation Based on Big Data Analytics: A Model Problem and Tool”. in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*: IEEE, 2017. DOI: 10.1109/seams.2017.20. URL: <https://doi.org/10.1109%2Fseams.2017.20>.