AAQC - Autonomous air quality control

Group 16 - Tobias Haller, Jonas Hofer, and Christian Müller

Service Computing Department, IAAS, University of Stuttgart

1 System Introduction

Increasing the well-being of people is an important topic of research. Especially for office spaces, when companies want to increase the efficiency of their employees. In general, with good air in terms of proper temperature, humidity and air purity, people feel more comfortable making them more productive. It can also be considered healthier, since previous research has shown that high levels of particulate matter in urban locations, where many offices are located, can be considered carcinogenic [1].

Another application could be retail stores where customers may stay longer, return at a later date or even recommend the store to their friends and family if they feel more comfortable, which may lead to higher revenue for the retail business.

The system we present in this document is designed to be operated in a singe office space. it can control the temperature, particulate matter and carbon dioxide (CO_2) levels in the room using a set of different sensors and actuators that allow the control of these parameters, like an air conditioner, and a ventilation system, which could also just be a window that can be opened and closed automatically. Our system was implemented to control a single room, since we wanted to build a functional basis that can be scaled up in the future, to be used for multiple rooms or even whole buildings.

Based upon this rough problem outline we performed a system analysis, defining functional and non-functional requirements in section 2. After the definition of the requirements we designed a system architecture, described in section 3, with the theoretical basis defined we started implementing the system itself, as well as a simulator to test the system without having a real world room to test in, as discussed in section 4. Finally, we conclude the report with potential expansion opportunities in section 5.

2 System Analysis

There are some core requirements and functionalities a user wants from this autonomous system. Given the described scenario we want to improve the situation of users by providing an autonomous way to control and adjust the air quality of a given room. In addition, we want to improve comfort of users by also regulating temperature or air purity of a given room based on environmental factors.

The core system functions will be separated into both functional and non-functional requirements according to ISO 25010. All requirements will be prioritized by - - / - / 0 / + and ++, where all requirements with + or ++ are considered to be mandatory for our project. All other requirements are desirable or optional.

2.1 Functional Requirements

Name FA-1
Title Autonomous temperature regulation
Description The system should be able to regulate the temperature of the room based on system information
Dependencies FA-8 FA-10
Priority ++

Name FA-2
Title Autonomous humidity regulation
Description The system should be able to regulate the humidity of the room based on system information
Dependencies FA-9 FA-10
Priority -

Name FA-3
Title Autonomous air quality regulation
Description The system should be able to regulate the air quality of the room based on system information. Particle count (Particulate matter) and CO_2 concentration are the basis for measuring the air quality.
Dependencies FA-6 FA-7 FA-10
Priority ++

Name FA-4
Title Night regulation
Description The System should reduce its functionality to a minimum during nighttime.
Dependencies FA-1 FA-2 FA-3
Priority 0

Name FA-5
Title Display system activity to User
Description The System should be able to show its activity and monitoring information to the user
Dependencies Everything else
Priority -

Name FA-6
Title Recognize stale air
Description The System should be able to recognize when the air has gone stale. This is based on the CO_2 concentration indoors.
Dependencies None
Priority ++

Name FA-7
Title Recognize increased particulate matter concentration
Description The System should be able to recognize when the air contains an increased concentration of particulate matter.
Dependencies None
Priority ++

Name FA-8
Title Monitor indoor and outdoor temperature
Description The System should be able to recognize the temperature indoors and outdoors of its operating location.
Dependencies None
Priority ++

Name FA-9
Title Monitor indoor and outdoor humidity
Description The System should be able to recognize the humidity indoors and outdoors of its operating location.
Dependencies None
Priority 0

Name FA-10
Title Detect human presence
Description The System should be able to detect human presence indoors.
Dependencies None
Priority -

2.2 Non-functional Requirements

ID NFA-1
Title Real world application
Description The system should be able to be easily transformed to a real world running smart home system.
Dependencies None
Priority -

ID NFA-2
Title Automation with minimal user intervention
Description The system should utilize AI-planning in order to automate its autonomous functionalities with minimal user intervention.
Dependencies None
Priority ++

ID NFA-3
Title Loosely coupled system
Description The system should be loosely coupled regarding its communication between devices.
Dependencies None
Priority ++

3 System Architecture Design

In the following we will specify our main components that execute our main functionalities. All components are separated into the following layers according to ISO/OSI model.

3.1 Presentation layer

The presentation layer consists of the user interface and the user itself. The (human) user interacts with our system via the user interface which is a GUI. The user can monitor the system as described in functional requirement FA-5.

3.2 Reasoning layer

The reasoning layer consists of the knowledge base, and the control unit. The knowledge base stores all the sensor data that is generated and stores information about the individual room with the state of all the actuators. It receives all its data from the context transformation component in the ubiquitous layer. The control unit interacts closely with the knowledge base, as it loads the measured sensor data from the knowledge base. Based on this data, the control unit decides based on AI-planning what actions it should take next. As soon as it has reached a conclusion and formulated a plan, it communicates this plan with the gateway in the ubiquitous layer.

3.3 Ubiquitous layer

The main component of the ubiquitous layer is the gateway which communicates between the physical layer devices and the components of the reasoning layer. It will process commands from the control unit of the reasoning layer directly in order to relay commands to the actuators of the system. Sensor data however will be sent to the context transformation component which will transform the raw data into a more precise and informative format the result is then communicated with the knowledge base of the reasoning layer.

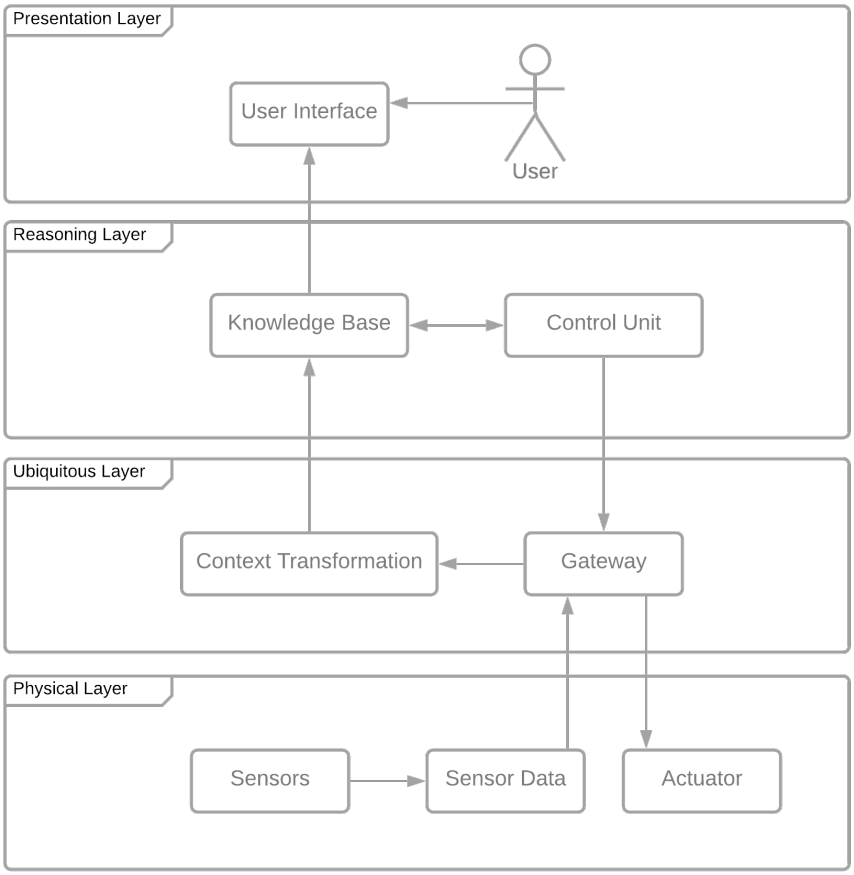

Fig. 1. The system design illustrated as a diagram

3.4 Physcial layer

There are two main components of the physical layer. Firstly the sensor which produces sensor data that will be sent to the gateway. Sensors will be used to satisfy functional requirements such as FA-8 and others. Secondly actuators will get instructions from the gateway and execute them as described in the functional requirements.

4 System Implementation

The system is composed of many different parts and components. This section will go into detail for all self implemented system components as well as describe how the components which are not implemented by us are integrated into the system. An overview of the systems components can be seen in fig. 2. The source code of the implementation of all seperate components can be found on GitHub[1].

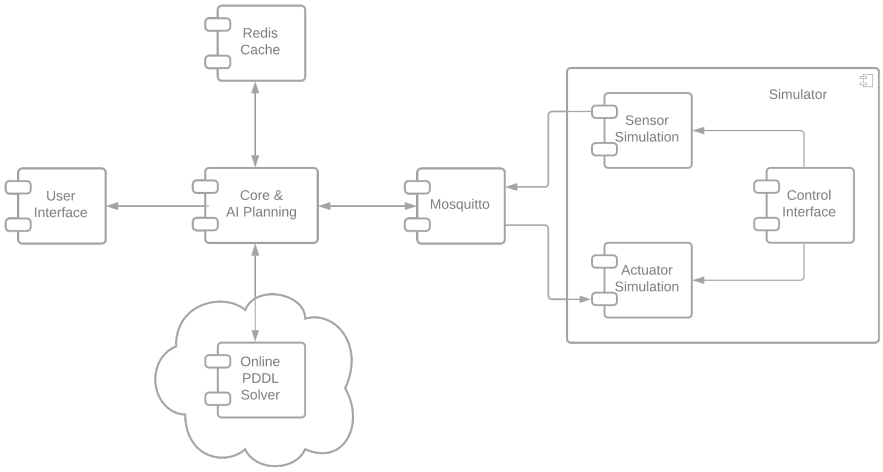

Fig. 2. Diagram of the system components

In our implementation, the system has six sensors and four actuators. The four actuators in our system or designated to activating or deactivating specific devices. In this implementation they are: 1) the ventilation 2) the heater 3) the air conditioner and 4) the air purifier. The sensors we use are: 1) a temperature sensor indoors 2) a temperature sensor outdoors 3) a particulate matter sensor indoors 4) a particulate matter sensor outdoors 5) a humidity sensor outdoors and 6) a CO_2 sensor indoors.

[1] `https://github.com/c-mueller/sc-iot-project`

4.1 Core

The application core is written using the C# programming language and consists of three different parts. Firstly The AI-planning which will be further elaborated in section 4.2. Additionally, a messaging endpoint component which has the job to receive incoming messages from the MQTT broker as well as send outgoing messages to this MQTT broker. This communication will be explained in detail in section 4.3. Furthermore, the application core has an HTTP API endpoint from which the User Interface, which is topic of section 4.5, can get user relevant information about the sensor measurements and current state of the systems actuators.

The communication inside the Core differs for the communication between the messaging endpoint and the AI-planning and the communication between the AI-planning and the HTTP API. The AI-planning component and HTTP API do not communicate directly with each other. Instead the AI-planning component saves some of the information about its current state, namely sensor data and actuator states, inside an external application state store. In our implementation this is realized using a Redis cache, but this could also be substituted with a regular relational database such as any SQL database, given an implementation has been provided based on our given interface. The HTTP API can then look up the saved information it needs and send it to the requester. The communication between the messaging endpoint and the AI-planning is more tightly coupled in contrast. These two components communicate directly through application intern function calls. On one hand the AI-planning does this when it has found changes in the current actuator state after it has finished one run of its planning routine. On the other hand the messaging endpoint does this periodically, so the AI-planning routine is not executed every time a new sensor measurement is sent to the MQTT broker.

4.2 AI Planning

The AI-planning routine is implemented using a mix between an external AI-planner and internal programming logic. It is the central part of the application core and therefore also implemented using C#.

The planning routine is initialized with the latest measurement of all sensors of the system in the so-called sensor context. All sensor measurements of this sensor context are uniquely identifiable for the AI-planning in order to know which physical location they come from.

Sensor context evaluation In a first step the AI-planning component evaluates this sensor context. There are different evaluation steps depending on the type of a sensor. The system has four different types of sensors which are the following: 1) Temperature 2) Humidity 3) Particulate matter and 4) CO_2.

Each sensor measure will be evaluated in relation to certain threshold values. Depending on the sensor type there are is either a threshold above or below which we want to take action or an acceptable value range above or below which we

want to take action. For example the evaluation of a temperature measure of 30°C indoors will be considered above a pre-defined temperature threshold for indoor rooms and therefore be too high. After the application finishes this sensor context evaluation it will have a sensor state in which all relations with the given thresholds of this sensor have been saved.

PDDL domain In this implementation we want a PDDL domain which is able to generate us any plan, which can be parsed and if necessary filtered or transformed to some extent, that lets us know the new state of our actuators, precisely if any of them are activated of deactivated. For this initialization, we can only use the previously evaluated sensor state as well as the current actuator state that we have saved inside the application state store.

Types The general Idea behind the Types is that we have both actuators and sensors at the core. Actuators are modelled as a `device` type and sensors as a `sensor` type. The sensors are then separated into which type of sensor they are, e.g. temperature, humidity, air purity or CO_2. Functionally, this would be sufficient to model the domain, however to improve readability and therefore make changes to the domain easier there are more types. Each Actuator, namely the ventilation, heater, air conditioner and air purifier have a type of their own. Additionally, Each sensor type has a type of their own that also has where this sensor is located, e.g. `temperature-in` which is of type temperature.

Predicates Most predicates of this domain will describe the sensor relation towards our given value or value range thresholds, as explained in section 4.2. This relation was already saved inside the sensor state and will be somewhat reflected in the predicated that were chosen to describe our PDDL domain. The following predicates currently exist in our domain:

- `(on ?d - device)`: This predicate is used to determine which actuator, and therefore the potential device which it controls, is currently on. If this predicate is true the actuator is activated and if it is false the actuator is deactivated.
- `(temperature-high ?t - temperature)`: This predicate together with the temperature-low predicate shows if a given temperature from a sensor is deemed not normal or acceptable. There need to be two predicated designated to modelling a threshold relation if the accepted value is a range. In this case, if the temperature-high predicate is true the temperature is too high, otherwise if it is false the temperature can be either normal or too low. Therefore, if both temperature-high is false and temperature-low is false, we know that the temperature for a given sensor is normal.
- `(temperature-low ?t - temperature)`: As mentioned, this predicate together with the temperature- high predicate shows if a given temperature from a sensor is seemed normal or acceptable. If this predicate is true the temperature is too low and if it is false the temperature is either normal or too high.

- `(humidity-high ?h - humidity)`: Unlike the temperature predicated where we needed multiple ones to model the sensor state, for humidity we are only interested if the humidity is too high. Therefore, if this predicate is true the humidity is too high otherwise if it is false the humidity is deemed acceptable.
- `(air-purity-bad ?a - air-purity)`: This predicate is used to determine the air purity, namely if there is too much particulate matter in the air. If this predicate is true we deem the air purity bad and if it is false the air purity is deemed acceptable.
- `(co2-level-emergency ?c - co2-level)`: This predicate indicates if the CO_2 level, in this case inside a given room, is either too high in which case this predicate is true or it is not too high and deemed acceptable in which case this predicate is false. This is a very problematic case and the system has to react more extremely to this than for other cases.

Actions The actions of our PDDL domain are exclusively related to activating or deactivating the actuators. There are these following actions:

- `activateVentilation`
- `deactivateVentilation`
- `activateHeater`
- `deactivateHeater`
- `activateAirConditioner`
- `deactivateAirConditioner`
- `activateAirPurifier`
- `deactivateAirPurifier`

The effect of these actions are mostly related to the `on` predicate in relation to each device of the system, e.g. one of the effects of `activateHeater` is `(on ?h)` and the effect of `deactivateHeater` is `(not(on ?h))`. These are all the effects of deactivate actions, however, activate actions also effect sensor related predicates, e.g. the other effects of the mentioned `activateHeater` action are also `(not(temperature-low ?ti))` and `(not(temperature-high ?ti))`, which say that the temperature inside a room are supposed to be neither too high nor too low. While these additional effects make parsing and filtering the plans necessary in some cases, which will be further elaborated in the next section 4.2, they also enable easier generation of problem files. This is the case because this way we can use a constant goal state, which is the same for any problem as well as make finding the initial states we need for a problem easier.

Parsing sensor and actuator state to PDDL problem The PDDL problem we want to generate depends on both the sensor state, which we evaluated previously, and the actuator state, which we received from the application state store. Additionally, the PDDL domain is the basis for any PDDL problem we want to generate.

The objects of any problem we generate is identical. Here we create an object for any actuator as well as all sensors of our system. For our problem, we need objects of the following types:

- `ventilation`
- `heater`
- `air-conditioner`
- `air-purifier`
- `temperature-in`
- `temperature-out`
- `humidity-out`
- `air-purity-in`
- `air-purity-out`
- `co2-level-in`

The initial states of a PDDL problem is the most interesting part, since these depend on the sensor measurements and the current actuator state. Actuator initial states are based on if they are active or not. For active actuators, each will have an `on` predicate added to the init of the problem. Due to the close world assumption of PDDL inactive actuators will not be part of the problem init section. The sensor state will also be parsed to init predicates of the PDDL problem. Here we will add the predicates as is described in section 4.2 to the problem init. Since we only have unwanted behavior as sensor predicates if there are no sensor predicates, we know the system is in a state where we do not need to activate any actuators. This is also the case for the actuators, since the best case for our system is that all sensor measurements are acceptable and all actuators are deactivated. This fact already describes what goal our PDDL problem will have. Furthermore, if any problem we create has no initial states, we also know that no action has to be taken without consulting any external AI-planner at all. This is due to the fact that such an initial state would mean all actuators are deactivated and any sensor measures acceptable values.

The goal of our problem is always the same. Find a plan for our current system, after which all actuators are deactivated and all sensor measurements inside the room our system runs for are acceptable. Precisely, we want the temperature to be neither too high nor too low, we do not want the air purity to be bad and there should not be too high of a CO_2 concentration inside the room. For our goal, the state of the outside sensors is irrelevant, since that is not something our system can change.

Create PDDL plan via external AI-planner In order to create any plan, our application has to interact with some external AI-planner which can create such a plan based on the given PDDL domain and problem. In order to stay flexible regarding which external AI-planner wants to be used, we provide an interface which has to be implemented. This interface gets a PDDL problem and returns a list of steps the plan consist of. If this list is either empty or null, we know that we don't have to do anything and can conclude the current AI-planning routine.

For our implementation we used an online PDDL solver[2]. This was done because this PDDL solver is very easy to use and integrate into the application. Additionally, other solutions we looked into are somewhat dated and have very limited or bad documentation on how to use them on top of other technical difficulties like, projects not being buildable at all and others. Here we can just send an HTTP request with the domain and problem file and this planner will return a response with the plan and some other meta information. For this requesting, we used Flurl[3].

Parse PDDL plan and execution Due to the way our PDDL domain and the PDDL problem is modeled, a given plan can not be takes completely at face value. We need to filter the given plan somewhat to get which actuators we have to activate or deactivate.

Since part of the goal of our problem is that all actuators are deactivated, a given plan will contain a step for activating as well as deactivating said actuator in the cases where in reality we only want to activate this actuator. This however can easily be filtered out by looking at which activation action have a paired deactivation step, since in practice we do not want to activate and deactivate an actuator in the same planning routine. This is not the case for deactivation, therefore, if there is only a deactivation step present in the plan we know that we really want to deactivate this actuator.

This filtering and parsing step will result in a new actuator state like the one we initially used before going into the PDDL problem parsing. This actuator state will have all states of the system's actuators, namely if they are activated or deactivated. If this new actuator state is the same as the one we used going into the planning, we know that nothing has changed, and we can successfully finish the AI-planning routine. If this is not the case, we will transform this information into the message format which the messaging endpoint wants and send such a message for each actuator for which we want to change its state. Finally, if there are changes in the actuator state, we will save the new actuator state inside the application state store, so it is up-to-date.

4.3 Communication

As mentioned in section 2.2, the communication between our components is ensured via a loosely coupled messaging protocol. For this purpose, we use the Mosquitto[4] message broker to send and receive JSON encoded data using the MQTT protocol. In the following we explain the communication in more detail. MQTT requires subscribing to so-called topics to receive messages instead of using a unique recipient address such as an IP address. The topics are designed to be extended for larger use cases. Their names start with an identifier for the location e.g. `room001` followed by `input` or `output` to specify the direction of

[2] `http://solver.planning.domains`
[3] `https://flurl.dev`
[4] `https://mosquitto.org/`

the topic, based on the view of the core application. In practice sensor data is always sent on an `input` topic and actuator messages are always sent on an `output` topic. Finally, the last part of the topic name defines the device or sensor the topic is assigned to, for example a topic for a temperature sensor ends with `/temperature`. The exception for this topic syntax are sensors placed outside of the building, here the direction is not defined, since we do not have any actuators outside. Some examples for valid topic names are:

- `room001/input/temperature` is the topic assigned to the temperature sensor of the room with the identifier `room001`.
- The outside humidity sensor publishes data on the topic `outdoors/humidity`.
- `room001/output/air-conditioning` is used to address the air conditioning unit of `room001`.

The messaging endpoint component of the core application subscribes to all available sensor topics to receive all measurements from sensors. Each sensor publishes and every actuator subscribes to its associated topic as described above. On one hand, this is necessary to ensure that the core component can receive the data from all loosely coupled sensors. On the other hand, this way all actuators can receive their instructions from the core application without talking to it directly.

The following listing illustrates a measurement message in JSON from the CO_2 sensor in `room001`. It contains the location of the sensor, the type of sensor, the time at which the measurement was made and the measured value as a floating point number.

```
{
    "location": "room001",
    "timestamp": "2021-07-05T07:30:01Z",
    "sensortype": "CO2",
    "value": 143
}
```

Listing 1.1. Example measurement message from a sensor in JSON format

Looking at the other direction, from the core application to the actuators, we just send a message regarding the actuation state of the respective device. When designing the initial idea of the system we thought of electrical relays or smart plugs as actuators that turn the device on or off depending on the messages from the core application. We therefore did not have any further configuration attributes like ventilation speed or target temperature in mind. Depending on the real world system adjustments in the core application may be necessary if the system should reconfigure these attributes too.

4.4 Simulation

Of course, it was not possible for us to build the system in person using actual senors and actuators. Therefore, we had to consider virtual i.e. simulated actuators and sensors instead. For this, we faced two initial options:

- Using an existing IoT simulation framework or platform
- Writing a simulation build for our purpose from the ground up

First we investigated potential simulation frameworks and platforms such as IoTIFY[5] or the Microsoft Azure IoT Device Simulation[6], however we could not identify a suitable solution that was not deprecated (Microsoft Azure IoT Device Simulation) or had a decent free version. We could not even identify a platform that had acceptable cost associated with it, however we did not investigate this far, since no one in our team wanted to spend money on this. The only remaining option was to build a simulation ourselves suitable for our use case from the ground up.

When implementing, we had several goals/requirements in mind:

1. Make the simulated devices as independent as possible while still allowing simple, mostly centralized, control of everything.
2. Make the simulator reconfigurable. For example, it should be easy to add a new sensor, change the topic of a device or adjust the default value of a sensor.
3. Make the simulator easy to deploy on either an x86-machine, a Raspberry Pi-like Single board computer or even the cloud

To fulfil the third goal, we could either use Docker, or a programming language that allows developers to produce assets without any external dependencies apart from their runtime, like the Java Runtime Environment. Our simulator was implemented using the Go programming language, since it produces a single statically linked binary with none or almost no external dependencies, depending on the compiler options chosen. Past personal experience has also shown that Go can be used to implement applications such as simple HTTP APIs or CLI utilities relatively quickly. To realize the simulation, the Gin web-framework[7] and the Paho MQTT Go Client[8] were used to implement the functionality.

To achieve the second goal, a YAML based configuration file was constructed that allows the configuration of the simulator in the expected manner. The following listing shows a simplified version of the configuration file for our simulator, in this sample the simulator only simulates sensors for indoor and outdoor temperature and an actuator for the ventilation. The meaning of the values in the configuration file are explained as comments in the listing.

[5] `https://iotify.io`
[6] `https://github.com/Azure/azure-iot-pcs-device-simulation`
[7] `https://github.com/gin-gonic/gin`
[8] `https://github.com/eclipse/paho.mqtt.golang`

```
# The HTTP port to listen on
http_port: 8080
# The host name/IP of the MQTT broker
mqtt_endpoint: 127.0.0.1
# The port the MQTT broker
mqtt_port: 1883
# The duration between simulated sensor measurements
sensor_update_interval: 1s
# The list of sensors to simulate
sensors:
  # The type of sensor supported values are:
  # Temperature, Humidity, CO2 and ParticulateMatter
- type: Temperature
  # The location identifier used by the simulator
  # internally, either Indoors or Outdoors
  place: Indoors
  # the unique name of the sensor, used for API access
  name: temperature-indoors
  # The initial sensor value to publish to the MQTT broker
  initial_value: 20
  # The name of the topic to publish on
  topic: room001/input/temperature
  # Model specific location identifier, this is
  # not used by the simulator itself it just
  # passes it through into the MQTT messages
  location: room001
- type: Temperature
  place: Outdoors
  name: temperature-outdoors
  initial_value: 20
  topic: outdoors/temperature
  location: outdoors
# The list of actuators to simulate
actuators:
  # The type of actuator, possible values:
  # Ventilation, AirConditioner, Heater, AirPurifier
- type: Ventilation
  # Same as Sensor
  name: ventilation
  # Same as Sensor
  topic: room001/output/ventilation
  # Same as Sensor
  location: room001
```

Listing 1.2. Sample configuration file for the simulator

Once the program is launched, the config file gets loaded and the simulated sensors and actuators are initialized according to the config file.

Every simulated device is kept independent of each other, meaning that they all have their own connection to the MQTT broker as well as their own thread/co-routine for publishing or subscribing. The only reason for keeping them within one application was the simplicity of deployment, and the ability to easily modify or access the device values and states from one central location without the need of distributed communication, like MQTT, gRPC or HTTP.

A sensor first publishes the value that has been set in the config file until the user changes the sensor value using the UI or the API. The current value gets published within a time interval defined in the configuration file.

The actuators work in a similar manner, but their value is not modified by the simulator UI, instead the value has to be set using MQTT messages from the topic the simulated actuator subscribes to.

Following this approach, the first goal has also been fulfilled.

Simulator User Interface Of course, the values for the virtual sensors have to be set somehow. We decided to implement this using a simple Web UI based on Angular. This UI shows all sensors and allows the user to set the new value that will be published by the simulator for the value of the specific sensor. The Web UI also shows the current state of the actuators, i.e. whether they are active or not.

The Web UI communicates with the simulator backend using an HTTP based REST-like API. The current value of an actuator or sensor is retrieved by polling an HTTP endpoint. We decided against the use of the more efficient alternative of a Publish-Subscribe approach using web-sockets, because we did not see any benefit, while the use of web-sockets or long polling may reduce the load on the backend if many users are connected to the UI, we only expect one or two concurrent users making high server load caused by redundant requests no concern to us. Doing it this way also made the implementation of both the UI and the backend API less complicated, which was way more important to us.

Once the simulation UI is compiled it only consists of static files, that could be served using any web server such as Apache httpd, assuming the path / URL to the API of the backend is set properly. In order to keep the deployment of the simulator as a whole simple, these static files are directly served using the simulator backend. The compiled JavaScript binaries of the Web UI are also included in the binary of the backend application. To achieve this, a Go library called packr was used. It loads all files in a previously defined directory into a go source file that defines the content of the directory as variables. packr then exposes these variables using an interface that is part of the Go standard library. Once the interface has been instantiated, it can be served through HTTP, making the directory or in our case the UI accessible over HTTP.

When a Go application is compiled it usually consists of a single file which is relatively large that includes everything, i.e. the UI, the Backend Code and the dependencies. This is comparable to a Fat JAR in Java.

Going into the real world The system in the current shape is pretty much usable in the real world, assuming of course, real sensors and actuators are used as we expect. In a real world system, one also has to implement the scripts for actuators and sensors, since the simulated ones do not fulfil the functional purpose of a real world device. The system will also become more distributed in the real world, since one device handling all six Sensors and four actuators is unlikely. The collection of data will therefore be handled in multiple locations.

4.5 User Interface

Similar to the simulation user interface we have also implemented a user interface for the core application. The goal of this user interface is to inform the user about the current state of the core application. It shows the current state of the measurements accumulated by the core application, as well as the expected state of the actuators. It shows the target values of the actuators and the actual values of the sensors, whereas the simulator ui shows the opposite.

Just like the simulator ui the Angular framework has been used to implement it, but unlike the simulator, the UI is not served via HTTP by an underlying application. Instead, it must be served separately, for example by using an external webserver.

5 Discussion and Conclusions

As we have seen, we developed a system that can control the temperature, CO_2 level and particulate matter level in a room. Improving the overall air quality using several quality improving actuators, like an air conditioner to reduce the temperature in a room, a heater to increase the temperature in a room, an air purifier to adjust the particulate matter level and the ventilation to bring the indoor levels to the outdoor levels.

To conclude, we have shown that an automatic system monitoring and adjusting air quality inside a room is possible and has high expendability for further additions. We could see some of the functional and non-functional requirements of such a system. All mandatory requirements as well as some more optional ones were implemented for this system by us. Our architecture was also based on common and proven IoT and smart city design principles. We have also shown how the implementation of such a system could be realized and how the AI-planning of such a system could look like as well as how this AI-planning can then be integrated into such a system. Additionally, a way to simulate sensors and actuators without the need of physical ones was elaborated. Furthermore, a way to integrate all different system components and for them to communicate with each other has been described.

As shown in this work, our current system should be able to operate in the real world, with some limitations, since the system in its current state can only turn actuators on or off. This limitation of course has impacts on the usability of the system, since the actuated devices can only operate at a preset speed / level.

This could impact the comfort of people in the room if the devices are operating on a high level causing unwanted noises like loud fan noises, for example. If the levels are set too low, the system may not operate at its full potential, which means it will take longer for the system to achieve the expected goal, potentially resulting in higher energy use.

Of course our system can be expanded with many more devices. Here are some example additions or improvements that could be made to the AAQC system:

- One could also monitor the indoor humidity levels and could then use a humidifier and a dehumidifier to keep the humidity level comfortable.
- The parameters for actuators could be expanded, to set the ventilation speed depending on the severity of the difference between indoors and outdoor parameters, for example.
- The system could also detect human absence to save energy if no one is using the room. This can save energy because an unused room could have different, less strict levels in comparison to the time when a human is present. For example, the temperature may go down to 15 °C and up to 30 °C, instead of 20 °C and 22 °C, if no one is in the room for a longer time.
- The time and date could also be taken into account, as an example the upper and lower thresholds may be changed based on the time or even the day of the week. This could also work depending on the time of the year, or maybe in combination with a weather forecast to know which times of the day would be best or worst for ventilation.
- There could also be an override for ventilating the room before anyone enters, e.g. early in the morning, to lower the CO_2 level to a minimum. As a result, the change of an CO_2 emergency where we have to ventilate could be lowered significantly, since this is unwanted on very hot or cold days.
- The system could be expanded, so that it may be able to control multiple rooms or even a whole building. However, when doing so, the usage of a local or self maintained AI-planner is mandatory.
- The system could also use a different AI-planner instead, for example one could use the `ff` (fast-forward) planner[9] locally instead of sending everything to the online planner we used. Doing so would also make the system completely independent of the internet.

The above list is by no means completed, there are many further additions and refinements one could think of adding to this system.

References

1. Pehnec, G., Jakovljević, I., Godec, R., Štrukil, Z.S., Žero, S., Huremović, J., Džepina, K.: Carcinogenic organic content of particulate matter at urban locations with different pollution sources. Science of The Total Environment **734**, 139414 (2020)

[9] `https://fai.cs.uni-saarland.de/hoffmann/ff.html`