

People Guidance System for Shopping Malls

Group 20: Marvin Bechtold, Maximilian Reichel, and Thomas Wangler

Service Computing Department, IAAS, University of Stuttgart `firstname.lastname@uni-stuttgart.de`

Abstract. Monitoring and guiding people in shopping malls are crucial to decrease the risk of infection with the Coronavirus. We propose a highly scalable and fault-tolerant system that combines the automatic measurement of the occupancy of each shop with an AI planning-based approach that guides customers in the shopping mall such that all customers are distributed equally among the shops regarding the preferences of the customers. Furthermore, displays in front of the shops can prohibit entry if the maximum number of people is exceeded. The measuring of the number of people is done by evaluating camera, Bluetooth, and WiFi data. Additionally, customers can view and evaluate the historical data for each shop to find time slots with less occupancy.

Keywords: Shopping mall guidance · Occupancy · Coronavirus

1 System Introduction

In the current context of the Coronavirus monitoring and controlling the room occupancy as well as the streams of people is an important task to reduce the infection rate. Therefore, it can be used to reopen facilities while minimizing the risks. For this purpose, our system counts the number of people in specific rooms, evaluates those numbers, and reacts to crowding by specifying alternative ways or blocking the entry to those rooms.

A concrete use case is a shopping mall. In shopping malls, crowds of people come together [3]. Without a guidance system, it is impossible to avoid gatherings at specific locations. In this context, our system regulates the number of customers visiting the shops in the mall. Each shop has a display showing the occupancy at this store. At a specified limit of concurrent customers, the shop gets closed until the number of customers decreases. Furthermore, the crowding in the whole shopping mall can be monitored. Based on the collected data, customers can be notified about times with an unusually high or low number of customers.

2 System Analysis

In this section, we describe what the users need and want from the system by introducing several user stories. The following user stories describe the functionality of the system:

- As a shopping mall owner, I need to control the number of people in the shopping mall and in particular in the shops so that I can reopen my shopping mall and stick to the current law.
- As a shop owner, I need to control the number of people so that I can reopen my shop.
- As a customer, I want to view the current number of customers at a shop so that I can avoid crowds of people.
- As a mall owner, I want to collect and monitor data about the number of customers and the shops they visit so that I can optimize the administration of the mall.
- As a mall owner, I want to automate the counting of people visiting a shop so that I can save costs.

- As a customer, I want to view historical data about the number of customers visiting a specific shop at a specific time so that I can schedule my buying in order to avoid crowds of people.
- As a shopping mall owner, I want that the system automatically generates recommendations for action for the customers so that it dynamically adapts to the current occupancy and distributes the customers.
- As a shop owner, I want to evenly distribute the customers over the time to flatten the peaks in rush hours so I can provide the best experience to my customers.
- As a customer, I want the possibility to get a time slot assigned from the shop website, so I can enter the shop without having to wait.

3 System Architecture Design

In this section, we give an overview of the system architecture. We separated the functionality of our system in different and independent components. Fig. 1 shows our components and the connection between them. The communication is implemented with message queues and HTTP calls. In the following, we outline the functionality of the logical layers in which we separated our components. For a detailed explanation about the implementation of the individual components see Section 4.

- Physical layer: WiFi access points and Bluetooth enabled devices are used to detect the presence of mobile phones. The number of people inside a building at a specific time can be estimated based on the number of connections from user devices to the access points [3]. Additionally, cameras can be used to improve the accuracy of the estimation. Displays in front of the shops are used to signal if a shop is crowded and, therefore, the customers have to wait or if they are allowed to enter the shop.
- Ubiquitous layer: The collected data of the WiFi and Bluetooth devices, as well as, the camera that is used in combination with image analysis is used to deduce the rough number of present customers at a specific area. Furthermore, human input is used to get the preferences of the customers.
- Reasoning layer: We use AI planning in this layer to generate recommendations for customers and plan their shopping tours based on the current occupancy and the preferences of the users.
- Presentation layer: Our system provides two user interfaces. Firstly, displays are installed to provide all visitors of the mall with information about the current occupancy and general recommendations for actions. The second user interface is tailored to the individual customer. Besides the public information, it also provides user-specific information to guide individual customers optimally. The customer can interact with the system via this user interface.

As sensors we have WiFi access points, Bluetooth enabled devices, cameras as well as user input with the customer web app. The sensor data is combined to estimate the number of customers in specific areas and stores. Shop displays and information for the customers via the web app are the actuators of our system. They regulate the number of customers at specific shops and assist customers at finding time slots with rather low occupancy.

4 System Implementation

This section gives an in-depth explanation of the implementation of the guidance system. The implementation can be found on <https://github.com/seedrix/SmartCities/>. A quick overview of the main system components is given in section 3. In the following the used tools and the implementation details of each system component, as well, as the communication mechanisms are described.

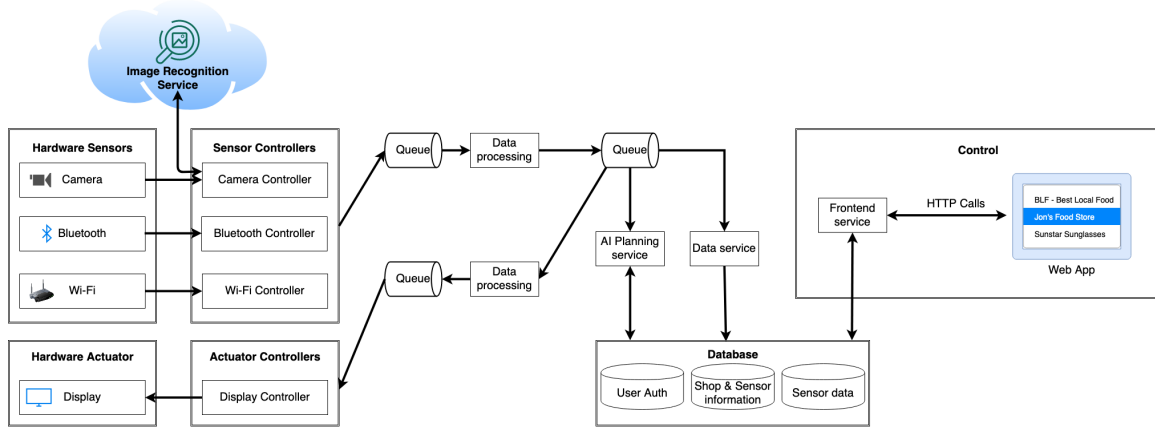


Fig. 1. Overview of the system architecture

4.1 Communication

For the implementation of the message queues of the system we used the Message Queuing Telemetry Transport (MQTT)¹ protocol. MQTT provides lightweight publish and subscribe messaging transport, optimized for high-latency and unreliable networks. To enable a fast development process, we used a public MQTT broker hosted by HiveMQ². In order to run our system in production, a private MQTT broker should be hosted.

To enable the communication between the web app and our system, an HTTP REST API is exposed by the frontend service. A detailed description of this API can be found in Section 4.6.

4.2 Sensors

To detect people, we used different types of sensors. Every sensor has a unique id, which is used to match the sensor with the shop it belongs to. The sensors publish their data into MQTT topics and all data is encoded as JSON. The topic path for each sensor is determined by its id. Every sensor publishes its data into some topic below `/sensors/{sensor id}/`, which is referenced as the sensor's topic in the following. An overview of all used topics is shown in Fig. 2. Analogously, also every shop has its own path in the topic hierarchy determined by its id: `/shops/{shop id}/`.

BLE Sensors Since most modern smartphones are able to communicate via Bluetooth low energy and most people carry a smartphone with them, we use these signals to detect people at the shops. Usually, devices only send signals, if they want to communicate with other devices. In our scenario, these other devices would be most likely smartwatches or headphones. So per default, we could not assume, that any customer is wearing a smartwatch or headphones. Just in time, Google and Apple released the Exposure Notification Framework³ for contact tracing. The protocol broadcasts

¹ <https://mqtt.org/>

² <https://www.hivemq.com/public-mqtt-broker/>

³ <https://www.apple.com/covid19/contacttracing>

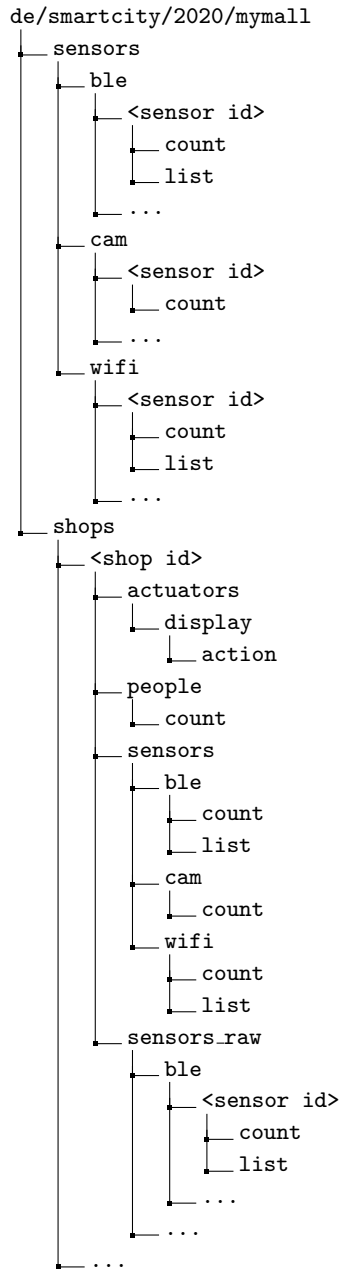


Fig. 2. MQTT topic hierarchy

periodically an id, which is changed every 15 minutes on average⁴, via Bluetooth low energy. Since the protocol communicates via Bluetooth low energy, we implemented our sensor to use these signals from the Exposure Notification Framework. Our BLE Sensor is designed to run on a Raspberry Pi with BLE capability as a python script. In python, we use the bluepy⁵ library to interface with the Bluetooth hardware. Since the Exposure Notification protocol was just released, there are no libraries to interact with it. Therefore, we implemented a parser by our self. In our python script, we first wait some time to receive the signals. If a signal is received, we extract the ID and add it to a set. After the scanning period, the recorded IDs are hashed for privacy reasons and published via MQTT. Afterward, the set is cleared and we start with the scanning again. We use a scanning period of 30 seconds. The data is published in two ways. The count of the elements in the set is published to the sensors `./count` topic. Secondly, the list with the hashed IDs is published to the sensors `./list` topic. We use this list in a post-processing step to remove duplicates (see Section 4.3). The ID of a BLE sensor starts with the letter 'b' followed by the mac address of the Bluetooth interface, so each BLE sensor uses a unique ID. In Fig. 4 you can see an example of the payload of the `./list` topic.

Cam Sensors Due to security reasons, cameras are omnipresent in shopping malls. They record the customers of the shops and cover nearly the whole shop area. We want to utilize the already existing security camera systems to detect and count people. To extract information from the video footage, we use an image recognition approach to recognize people in the video footage. In order to keep the effort low and develop a prototype quickly, we used the cloud service Amazon Rekognition⁶ for the image recognition task. Furthermore, this ensures accurate results in a scalable manner. The basic idea of this sensor is to extract single pictures from the video footage of the cameras in a predefined interval. Afterward, the pictures are sent to the AWS Rekognition service to label and count the people. To increase the simplicity when deploying our system in a real environment, our sensor takes all images into account that are placed in a certain folder. The only modification needed for an existing camera system is a script that periodically extracts images from the video stream of the camera and stores them in the folder of our sensor. When the sensor detects a new image in the folder, it uploads the image to Amazon S3⁷ and then triggers the Amazon Rekognition service to label the picture. Our sensor receives the resulting labels and extracts the people count from the result. This data is then published in the topic `/sensors/cam/{some sensor id}/count`.

WiFi Sensors Many shopping malls provide free WiFi to their customers. Also, the customers require an internet connection for their phones, to be able to interact with our web app. In our approach, we used a Raspberry Pi to act as an access point to which the customers can connect to. For this, we used the software RaspAP⁸. RaspAP provides an easy administration interface to set things up. In our configuration, each Pi serves as a DHCP server to assign IP addresses from a unique pool to the WiFi clients. Our python script uses shell commands to periodically extract a list of the connected WiFi clients. For this period, we used 30 seconds. The extracted data is published in two ways: The number of connected clients is published to the sensors `./count` topic. Since the MAC address of a device is unique, we can use it to identify a client across multiple access points. To increase the privacy of the customers, we are using the hashed value of the MAC address for

⁴ <https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ExposureNotification-BluetoothSpecificationv1.2.pdf>

⁵ <https://github.com/IanHarvey/bluepy>

⁶ <https://aws.amazon.com/rekognition>

⁷ <https://aws.amazon.com/s3/>

⁸ <https://raspap.com>

further processing. We generate a map with the device IP addresses as keys and the hashed MACs as values and publish this map to sensors `/list` topic. We use this map in the later processing to remove duplicates (see Section 4.3) in the case that a device switched to another AP inside the shop. Moreover, this data allows us to implement a mapping from a WiFi client to a user's account, so we could locate users across the mall. However, this remains for future work (see Section 5). The ID of a WiFi sensor starts with the letter 'w' followed by the mac address of the WiFi interface which is used for the AP. Therefore, each WiFi sensor uses a unique ID. In this approach, we can only detect phones which are connected to our WiFi AP. There are alternative ways that work without the requirement that the customer must connect their phone to our AP. One approach is described by Georgievska et al. [1].

4.3 Sensor Data Processing

The raw sensor data from each sensor is not very meaningful by itself. Therefore, we do some post-processing to the sensor data to add context and aggregate the values from multiple sensors to one value of a virtual sensor. All components described in this section are written in python and communicate only via MQTT with each other. This increases the flexibility and the loose-coupling of the system since each component could easily be swapped out and other components would be able to get data at any processing step in the processing pipeline. An overview of the post-processing steps is shown in Fig. 3. At the first stage, the sensor data is enhanced with the information to which shop the sensor belongs to. This is done by the **ShopMapper** component. In the second stage, the data is aggregated for each shop and sensor type to a virtual sensor, e.g. data from all BLE sensors of one shop is combined to one overall virtual BLE sensor value for this shop. This is done by the respective **Aggregator** component for the sensor type. In the third stage, we combine the values for all sensor types per shop to calculate one value for the virtual people sensor which represents the estimated people at this shop. This is done by the **PeopleAggregator** component.

Sensor Shop Mapping The **ShopMapper** component is subscribed to the general `/sensors/` topic and redirects the messages from the sensors to the corresponding shops. To achieve this, the **ShopMapper** retrieves information about all shops from the database at startup. A mapping in which sensors and shops correspond to each other can be extracted from this data. Concretely, every message to the `/sensors/ble/{some sensor id}/list` topic would be published to the `/shops/{some shop id}/sensors_raw/ble/{some sensor id}/list` topic of the appropriate shop in which the sensor is located. This is done analogously for the WiFi and cam sensors and the `count` topic. Additionally, the message payload is enhanced with a `shop_id` attribute to ease to further processing. Fig. 4 shows an example with payloads.

BLE Sensor Data Aggregation The **BleAggregator** component combines the data from all BLE sensors per shop and acts as a virtual BLE sensor per shop. It is subscribed to `/shops/+ /sensors_raw/ble/+ /list`, so one instance of the component can handle the BLE sensors of all shops. The **BleAggregator** stores the last received message from all BLE sensors per shop. If a new message is received, the data of the messages is aggregated and published into the `/shops/{some shop id}/sensors/ble/count` and `/shops/{some shop id}/sensors/ble/list` topics of the shop. Since the Bluetooth signal of a device could be received by multiple sensors, duplicates can appear in the recorded IDs. To tackle this issue, we add all distinct IDs to a set and publish it to the `ble/list` topic. Afterward, the number of elements in the set is published to the `ble/count` topic. Fig. 4 shows an example with some payloads.

Cam Sensor Data Aggregation The **CamAggregator** component combines the data from all cam sensors per shop and acts as a virtual cam sensor per shop. It is subscribed to `/shops/+ /sensors_raw /cam/+ /count`. It sums up the counts from the cameras of a shop and publishes the result to `/shops/{some shop id} /sensors /cam /count`.

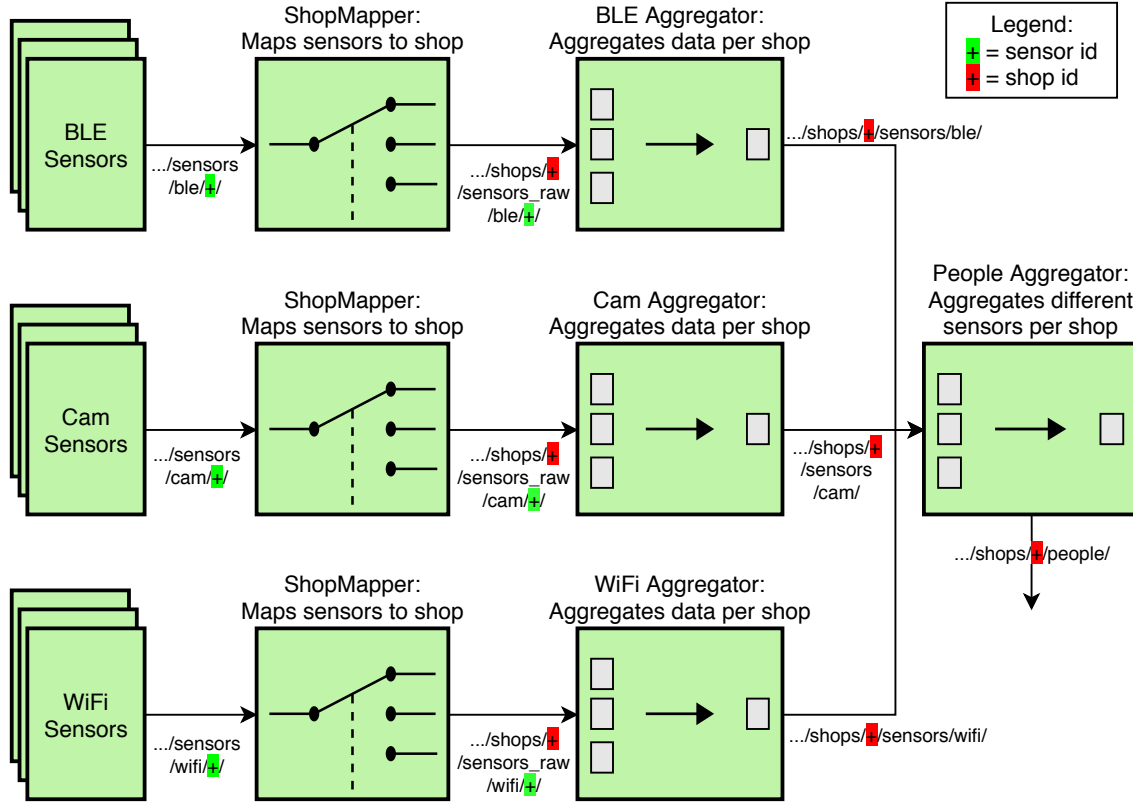


Fig. 3. Sensor data post-processing

WiFi Sensor Data Aggregation The **WiFiAggregator** component is very similar to the **BleAggregator** component and acts as a virtual WiFi sensor per shop. It is subscribed to `/shops/+ /sensors_raw /wifi/+ /list`. The main difference is that the **WiFiAggregator** works on maps instead of lists. It aggregates the maps in a way that every value in the resulting map is unique. This treats the case that a device was recorded by multiple sensors which can happen if the device switched between access points inside a shop.

Sensor Data Aggregation per Shop The **PeopleAggregator** component combines the data of the BLE, camera, and WiFi data per shop. So it acts as the virtual people sensor per shop. It is subscribed to `/shops/+ /sensors/+ /count` and calculates the mean of the count-values from all the sensor types of the shop. This value is published to `/shops/{some shop id} /people /count`. Fig. 4

shows an example with a message to this topic. To ease the monitoring, this message also contains information about the number and types of the aggregated sensors used to calculate the value.

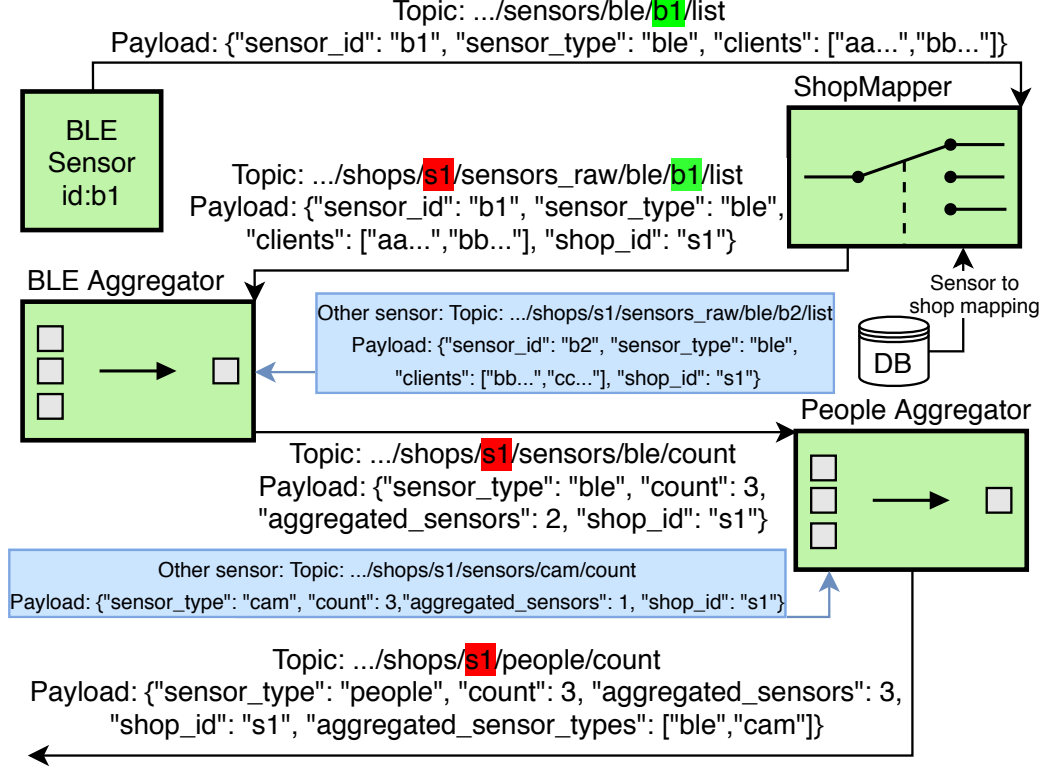


Fig. 4. Example message flow of BLE sensor data processing with message payloads

4.4 AI Planning

The AI planning service is the core of our system. It computes plans for all customers such that the people are evenly distributed among the shops in the shopping mall regarding the capacities of each shop and the requests of the customers. As the AI planning solver we use Metric-FF⁹ which is an extension of the FF¹⁰ planner to support numerical functionality. We chose this solver because it supports level 2 of the PDDL description [2]. This is required for our system because for the distribution of the customers we need to be able to compare numerical values in the PDDL problem file. Furthermore, Metric-FF is highly performant [2].

The AI planning service queries periodically the users' shop preferences and current and maximum shop utilization from the database and starts the AI planner on this data. Afterward, the next shop attribute of each user is updated in the database accordingly to the decision of the planner.

⁹ <https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html>

¹⁰ <https://fai.cs.uni-saarland.de/hoffmann/ff.html>

Domain File Listing 1.1 shows the PDDL problem file. We have the objects list and shop. A list describes the shops a customer wants to visit and is associated with a user. At-shop is the predicate that describes the recommendation for the next shop the customer should visit. This is the main part of the computed plan. The predicate list-set describes if a shop is chosen for a specific list. At the end of the computation, each list has to have an associated shop. Shop-option describes the shop preferences of the user. For each selected shop of a user, there is a shop-option combining this user and the shop. The action choose determines a shop for a user. This action is not possible if for this user a shop is already chosen or if the shop is not one of the user's preferences. The effect of this action is that the predicates list-set and at-shop are set accordingly as well as the function people-at-shop for the chosen shop is increased by one.

Listing 1.1. PDDL problem file

```
(define (domain Guidance)
  (:requirements [:typing] [:adl])
  (:types shop list      object)
  (:predicates
    (at-shop ?x - list ?y - shop)
    (list-set ?x - list)
    (shop-option ?x - list ?y - shop)
  )

  (:functions (people-at-shop ?x - shop) - number)

  (:action choose
    :parameters (?list -list ?shop -shop)
    :precondition (and (not (list-set ?list))
                       (shop-option ?list ?shop))
    :effect (and (list-set ?list)
                 (at-shop ?list ?shop)
                 (increase (people-at-shop ?shop) 1)
    )
  )
)
```

Problem File The problem file has an object of type list for each active user who selected a list of requested shops he wants to visit. Moreover, the shops in the shopping mall are defined. The initial value for the function people-at-shop is set to the number of people the sensors captured in that shop (see Section 4.2). The goal is that all list-set holds for all lists and the function people-at-shop is lower than a specified limit. This limit is initialized with the maximum number of people that can visit the shop at the same time. If the planning problem does not have a solution, the limit is increased iteratively until a solution is found. In this case, some customers must wait before they can visit a shop which is displayed by monitors in front of each shop (see Section 4.8).

At first glance, the problem that needs to be solved might rather look easy to solve. However, trivial approaches like selecting one of the shops that are requested and have still capacity do not lead to the best solutions and are in general inaccurate. Solving the problem with AI planning and

the described construction of the problem produces the optimal solution and has a high performance as long as the number of shops is not uncommonly high.

4.5 Database

We use a MongoDB¹¹ database for the persistent storage of our data. We have three different types of data that we store in databases:

1. The authentication data for user which contains the email and the hash value of the password
2. The information about shops and sensors and the mapping between them
3. Sensor data which is published over MQTT topics

We use the data service component to automatically store MQTT messages from specific topics to our database. This is relevant to provide access to historical data for other components of our system. Our implementation is based on <https://github.com/David-Lor/MQTT2MongoDB>.

4.6 Frontend Service

The frontend service exposes a REST API to interact with our system. To create the API, we used the framework Flask¹² in combination with the extension Flask-RESTful¹³.

The frontend service is divided into three sub-APIs: Shops, User, and Auth. In the following, all three are described.

Shops-API The Shops-API handles all requests regarding the information of the shops.

/shops/all

GET

Description: Returns a list of all shops

Postcondition: Content-Type: application/json

Status code: 200 OK

500 Internal Server Error

¹¹ <https://www.mongodb.com/>

¹² <https://github.com/pallets/flask>

¹³ <https://github.com/flask-restful/flask-restful>

/shops/shop/<string:shop_id>

GET

Description: Returns specific information for the requested shop.

Precondition: **shop_id** is a valid ID for a shop in the database

Postcondition: Content-Type: application/json

Status code: 200 OK
 404 Not Found
 500 Internal Server Error

PUT

Description: Creates or updates a shop

Precondition: Content-Type: application/json

body contains a valid description of a shop

Body: { "max_people1" : 2, "sensors"
 : ["bB8:27:EB:2A:C9:E6",
 "wb8:27:eb:d5:36:19", "ccamera0"
], "shop_name" : "BLF -
 Best Local Food", "logo" :
 "data:image/svg+xml;base64,PD94bW..."
 }

Status code: 200 OK
 400 Bad Request
 500 Internal Server Error

/shops/people/<string:shop_id>

GET

Description: Returns the current number of people at the requested shop

Precondition: **shop_id** is a valid ID for a shop in the database

Postcondition: Content-Type: application/json

Status code: 200 OK
 404 Not Found
 500 Internal Server Error

`/shops/people/<string:shop_id>/<int:timestamp>`

GET

Description: Returns all people data for the given shop after the requested timestampPrecondition: **shop_id** is a valid ID for a shop in the databas
timestamp is a valid unix time timestamp

Postcondition: Content-Type: application/json

Status code: 200 OK
404 Not Found
500 Internal Server Error

Auth-API The Auth-API handles user authentication. It enables the user to sign up and to log in to our application. For authentication purposes, a JSON Web Token (JWT) is generated. The whole state is included in the JWT which saves us to manage the authentication tokens on the server-side. The JWT expires after one day which is suitable for our use case.

`/auth/signup`

POST

Description: Creates a new user account with the given email and passwordPrecondition: body contains an email and password
the email address is not already in use
Content-Type: application/jsonPostcondition: send JWT token
create userBody: { "email": "test@email.com", "password":
"test1234" }Status code: 200 OK
400 Bad Request
500 Internal Server Error

/auth/login**POST**

Description: Log in a registered userPrecondition: body contains an email and password
the user exists in our database
email and password are correct
Content-Type: application/json

Postcondition: send JWT token

Body: { "email": "test@email.com", "password":
"test1234" }Status code: 200 OK
400 Bad Request
500 Internal Server Error

User-API The User-API handles all user-specific tasks. A JWT must be attached to all requests to this API for authentication purposes.

/user/next_shop**GET**

Description: Returns the recommended next shop for the user

Precondition: JWT

Postcondition: Content-Type: application/json

Status code: 200 OK
401 Unauthorized
500 Internal Server Error

`/user/shops`

GET

Description: Returns the list of selected shop for the user

Precondition: JWT

Postcondition: Content-Type: application/json

Status code: 200 OK
 401 Unauthorized
 500 Internal Server Error

PUT

Description: Sets the list of selected shops for the user

Precondition: JWT

Content-Type: application/json
 body contains valid list of shops

Body: ["shop1", "shop2"]

Status code: 200 OK
 401 Unauthorized
 500 Internal Server Error

`/user/shops/<string:shop_id>`

DELETE

Description: Deletes the shop from the list of selected shop, if the shop was on the list

Precondition: JWT

Status code: 200 OK
 401 Unauthorized
 500 Internal Server Error

4.7 Frontend

A web app is used to interact with our guidance system. Furthermore, users are notified which shop should be visited next. We used the Angular Framework¹⁴ to implement our frontend. Additionally,

¹⁴ <https://angular.io/>

our implementation uses Angular Material for the components design and MDBBootstrap Free¹⁵ for chart visualizations which is licensed under the MIT license¹⁶. Using a component library leads to a consistent look of the user interface. The frontend gets the latest data by HTTP requests to the frontend service (see Section 4.6). In the following, we describe the frontend and its functionality. The navigation is done with a side menu to increase the usability on mobile devices (see Fig. 5).

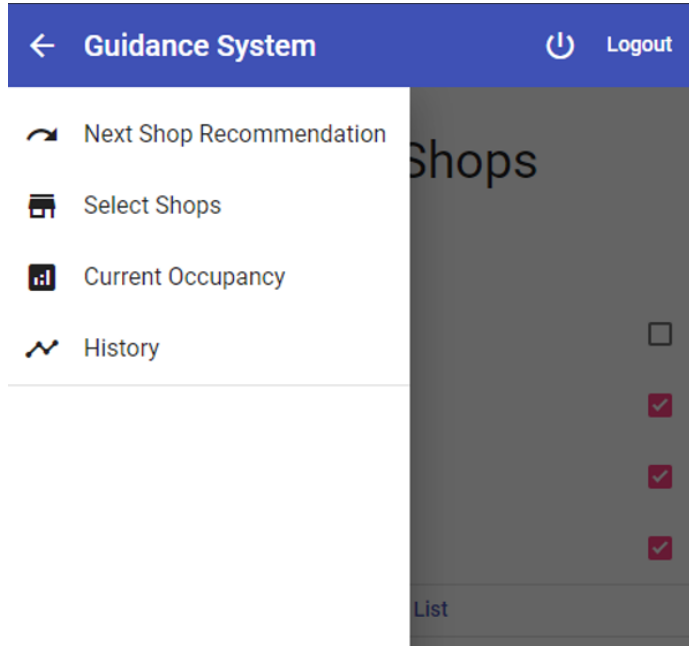


Fig. 5. Side menu

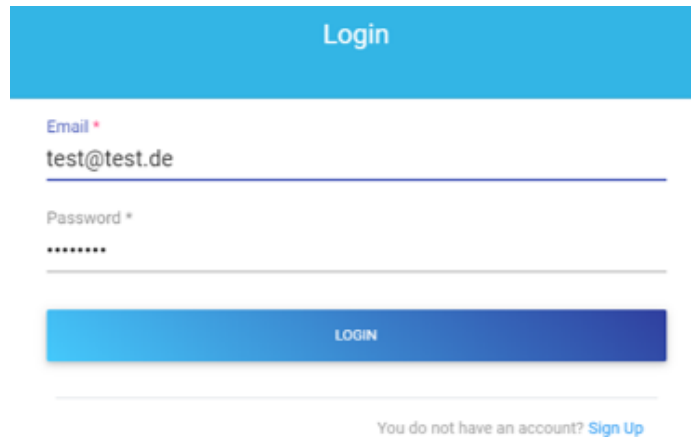
User Management The web app contains a user management. Users can sign up and log in to their accounts by specifying their e-mail address and a password (see Fig. 6). The form automatically checks if the given input is syntactically valid. The authentication is done with a bearer token¹⁷. The token is returned from the frontend service and saved in the local browser storage. HTTP-Requests that are sent to the backend are then complemented with the token. The user management enables the use of the application on several devices.

Shop Selection The frontend contains a page that shows all shops in the shopping mall and allows the user to select a subset that he wants to visit during his purchase at the mall. The data is sent to the frontend service with an HTTP-Post message. Afterward, the recommended shop for the user that is computed with the AI planner (see Section 4.4) considering the user lists, as well as the current occupancy rates are shown.

¹⁵ <https://mdbbootstrap.com/docs/angular/>

¹⁶ <https://mdbbootstrap.com/general/license/>

¹⁷ <https://oauth.net/2/bearer-tokens/>



The image shows a login page with a blue header bar containing the word "Login". Below the header, there are two input fields: "Email *" with the value "test@test.de" and "Password *" with masked characters "*****". A blue "LOGIN" button is positioned below the password field. At the bottom, there is a link that says "You do not have an account? [Sign Up](#)".

Fig. 6. Login page

Data Visualization The database contains the number of customers at each shop. This occupancy rate can be displayed in a bar chart for each shop next to the maximum number of customers for that shop. Furthermore, the historical data can be shown in a line chart (see Fig. 7). This is useful for customers to analyze the times at which the shopping mall in general or the shops they want to visit are less frequented and to avoid crowds of customers, therefore.

4.8 Displays

The **ShopDisplay** components are used to control the number of people in the shops. A **ShopDisplay** is a display in front of the shop which shows to the customer if he is allowed to enter the shop or not. We implemented the shop displays as an HTML page. The page connects via WebSocket to the MQTT broker and subscribes to the *actuators/display/action* topic of the particular shop. The page is stateless and the shop is set via the URI fragment. For example, the page *display.html#shop1* will subscribe to the topic */shops/shop1/actuators/display/action*. The JSON messages in this topic contain the following attributes: **symbol**, **color** and **message**. **color** defines the background color of the page, **message** contains the message which should be displayed, and **symbol** contains a base64 encoded image, which will be displayed below the message text. This allows us to display highly customizable messages to the customers. In fact, the message attribute could also contain HTML content which would be rendered on the page.

Display Controller The **DisplayController** component is used to control the **ShopDisplay**. The **DisplayController** retrieves the maximum number of allowed people per shop from the database and listens to the *people/count* topic of the shops. If the people count is above 95% of the maximum shop capacity, the **DisplayController** will publish a message with the message text "Don't enter! Please wait", red background and an appropriate symbol to the */shops/{some shop id}/actuators/display/action* topic of the shop. If the utilized capacity is below this threshold, it will publish a message to this topic, which will encourage the customers to enter the shop.

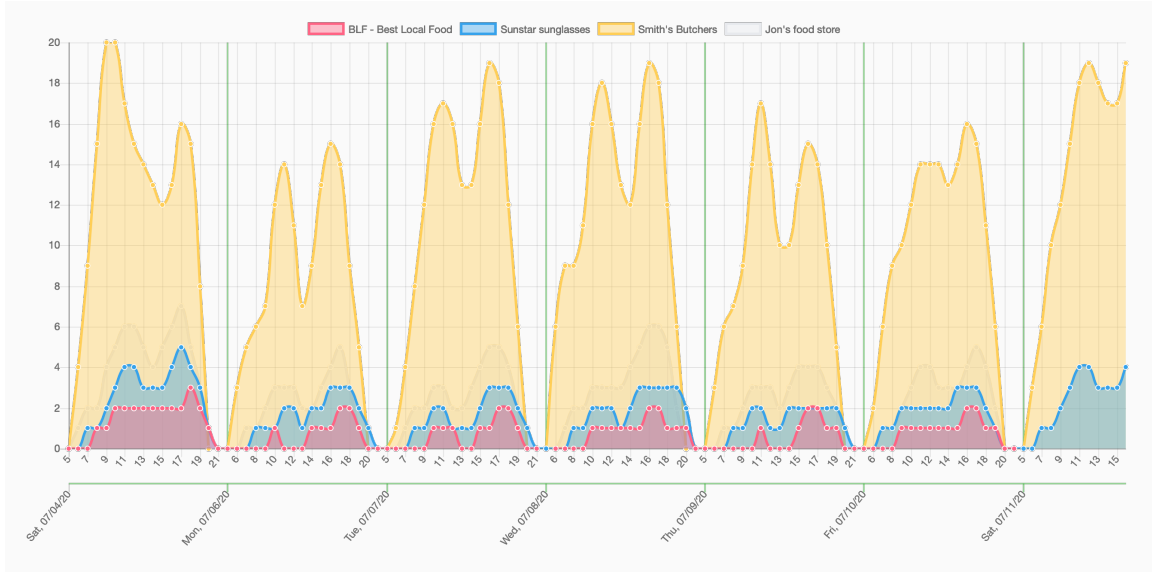


Fig. 7. Visualization of the history for one week

4.9 Test Data Generation

Since we do not operate a shopping mall, we are not able to record real data to test the system. In order to tackle this issue, we implemented two ways of data generation: Live sensor data and historical sensor data. To generate live sensor data, we implemented a dashboard that acts as simulated sensors and provides some simple monitoring features. To obtain historical data, we wrote a script to extract data from Google's *Popular times* feature and insert it into our database.

Simulated Sensors To test the system without the need of a bunch of Raspberry Pis and real people, we wrote a sensor dashboard where we can set the values of simulated sensors for each shop. Also, it provides us with monitoring features that are helpful for debugging. The dashboard is written as a web app with HTML and JavaScript. It uses MQTT to publish sensor data to the sensor topics like the real sensors would do. The values can be set manually or be copied from an array of values with some delay to simulate sensors over time. Additionally, the dashboard shows the values of our virtual aggregated sensors per shop.

Historical Test Data To gather historical data, we used data from Google Maps' *Popular times*¹⁸ feature. This feature shows how busy a particular shop is per hour and weekday. In order to extract this data, we wrote a JavaScript snippet to gather the HTML data of the page. A python script then uses this data and maps it to the capacity of a shop and writes it to our database in the same way the data of the virtual people sensor would be written to the database.

¹⁸ <https://support.google.com/business/answer/6263531?hl=en>

5 Discussion and Conclusions

The implemented guidance system can be used to guide customers in a shopping mall with the goal to distribute them equally among the individual shops. This results in less waiting time for the customers as well as less crowding in the shops. The system is highly scalable and fault-tolerant due to the usage of the messaging protocol MQTT. Moreover, our responsive web app ensures usability.

Future work is the improvement of the accuracy of the sensors by using machine learning. In this context, the various sensors can be trained with annotated data from the environment in which the system runs. This can increase the accuracy of the people estimations. Another aspect of future work is to extend the planning service to consider historical data to improve the recommendations. Furthermore, this would allow us to give guiding recommendations for the future. Instead of suggesting only the next shop a user should visit, the system could output a list of shops. Additionally, the mapping from WIFI clients to users could be used to track the users automatically. This could be used to increase the usability of the system because the input that a user left a shop could be omitted in this case. To conclude, one can say that our system shows the feasibility and the usefulness of a guidance system in shopping malls.

References

1. Georgievska, S., Rutten, P., Amoraal, J., Ranguelova, E., Bakhshi, R., de Vries, B.L., Lees, M., Klous, S.: Detecting high indoor crowd density with wi-fi localization: a statistical mechanics approach. *Journal of Big Data* **6**(1) (mar 2019). <https://doi.org/10.1186/s40537-019-0194-3>
2. Hoffmann, J.: The metric-ff planning system: Translating “ignoring delete lists” to numeric state variables. *Journal of artificial intelligence research* **20**, 291–341 (2003)
3. Nguyen, C.T., Saputra, Y.M., Van Huynh, N., Nguyen, N.T., Khoa, T.V., Tuan, B.M., Nguyen, D.N., Hoang, D.T., Vu, T.X., Dutkiewicz, E., et al.: Enabling and emerging technologies for social distancing: A comprehensive survey

All links were last followed on July 11, 2020.