Final Report

Server Program for Retail RFID System

Mike Beijen
Kevin Chong
Callum Holland
Glenn Keller

Delft University of Technology

TU Delft Delft University of Technology

Challenge the future

FINAL REPORT

SERVER PROGRAM FOR RETAIL RFID SYSTEM

by

Mike Beijen
Kevin Chong
Callum Holland
Glenn Keller

in partial fulfillment of the requirements for the degree of

Bachelor of Science
Computer Science and Engineering

at the Delft University of Technology,
to be defended publicly on Wednesday, July 3, 2019, at 14:00.

Coach:	Dr. M. Finavaro Aniche,	TU Delft
Client:	Dr. P. Pawełczak,	TU Delft
BEP Coordinators:	Dr. H. Wang,	TU Delft
	Ir. O. Visser,	TU Delft

TU Delft Delft University of Technology

PREFACE

This is the report for the Bachelor End Project (2019), which is a required course for the BSc Computer Science and Engineering curriculum at the Delft University of Technology. The duration of the project was eleven weeks, in which the goal was to design and develop an application for the Embedded and Networked System group under the supervision of Dr. Przemysław Pawełczak. The project entails developing a retail environment RFID system to allow further research to be done on this topic.

We want to thank the people who supported us during the project. Thanks go out to our supervisor Dr. Przemyslaw Pawełczak for offering this project, his great devotion, and excellent guiding. We would also like to thank our coach Dr. Mauricio Aniche, for his valuable feedback. Lastly, we would like to thank Thijmen Ketel for his helpful clarifications.

M.F. Beijen
K. Chong
C.R. Holland
G. Keller

Delft, June 2019

SUMMARY

The Embedded and Networked Systems research group at TU Delft is researching RFID technology, one of the ongoing research topics is related to allow messages to be encoded on the information received by an RFID reader. Their solution would allow the costs of using RFID systems to be lowered, due to the removal of the handheld RFID readers. These handheld readers would instead be replaced by the smartphone of a user. However, to fully showcase their envisioned system, the underlying system would need to be designed and developed.

The solution developed allows information from the RFID tags to be stored in a database on the server and changes to this information can be initiated from the Android application without knowledge of what the EPC of the tag is beforehand. The complete system has been developed with scalability and maintainability in mind. The system was thoroughly tested using unit testing and integration testing on the server and the Android application, as well as manual end-to-end testing with the complete system. To evaluate the final system, several tests were performed that were able to find out several limitations of the systems, which require further attention in later research.

Contents

1

INTRODUCTION

Radio Frequency Identification (RFID) is a technology that enables data from tags to be read from a distance. RFID is used in many varying settings, such as in retail environments. In a store, the tags can be attached to each product that needs to be sold, whereas the reader can be either a fixed reader that is placed somewhere in the store or a handheld reader that can be carried by the employees of the store. Both readers can be used at the same time for different purposes. The fixed reader allows for an overview of the whole store. While the handheld reader allows an employee to inspect a specific tag. The handheld readers allow for more flexibility and are rather expensive. The researchers of the Embedded and Networked Systems group at TU Delft have researched this topic and have found a possible alternative solution using an Android device, which would remove the need for these handheld readers.

However, the envisioned RFID system requires a unique software solution that differs from the available RFID systems on the market. As the communication between the RFID reader and the RFID tags are one-way channels the Android device would not be able to obtain the information on electronic product codes (EPCs) directly from this channel. To allow for further research to be done on this topic as well as to showcase their envisioned RFID system, the Embedded and Networked Systems research group at the Delft University of Technology has requested a system to be designed and developed to allow communication of tag information between the RFID reader and an Android device.

In this project, the architecture for the envisioned system is designed and developed. An API is made for the server to allow for communication with the database and cache. An Android application is developed to demonstrate the capabilities of the system. A connection was established between the Android application and the microcontroller to allow for messages to be transferred. The setup for the RFID reader is connected to the server to allow for storing the information on the database. A dashboard that can be accessed from a web page was created to make the data stored more accessible. Several options to add or modify the data stored were included to improve the usability of the system.

2

PROBLEM DEFINITION

The Embedded and Networked Systems (ENS) group [1] is a research group at TU Delft. The research of this group concerns the software side of Embedded Systems. One of the ongoing research within this group is in removing the need for handheld RFID readers. One of the use cases for RFID systems is in retail environments (i.e., shops), where every product will get a tag to identify the product.

Stocktaking in a store can be done automatically using the unique ID of each tag. A database will be used to link each unique ID with the information on a product (e.g. name, price, expiration date). This information is generally added to the database when a product first arrives at a store, to modify this information afterwards an employee would need to either bring the products to a reader that can single out a specific tag or use a handheld RFID reader. The handheld reader will interact with a tag and obtain information (e.g. unique id for a tag, name of the product, price of the product) from this tag. The user can then modify this information.

The need for handheld readers could be removed by only using a fixed RFID reader that can relay information about tags back to an Android device. The idea is to have a microcontroller attached to the Android device. This microcontroller will be used to encode a message from the Android device to piggyback it on the messages received by the fixed RFID reader. These messages will contain an operation that needs to be performed by the server on the accompanying tag. The server can then provide feedback to the Android device to inform whether the process was successful or not.

To achieve this goal, new software needs to be designed and developed to allow for communication between the different components in the envisioned system. This product will help with further research into this new approach. During the development, the RFID system was envisioned to be used in a retail environment, despite different use cases (e.g., warehouse) being possible as well. This is because the primary purpose of the tool is to help with research and serve as a demo for the research of the ENS group, for this purpose a retail environment would be more fitting as the reader would be relatively more familiar with the setting of a store compared to a warehouse.

2.1. REQUIREMENTS

The final product will consist of several components that will either be developed from scratch or be adjusted to fit the need of the project, these components are:

- A server
- An Android application
- A microcontroller (e.g. Arduino)
- An RFID Reader

[1] www.ens.ewi.tudelft.nl

Each component will have its own set of requirements that will need to be fulfilled. For the final product to be used by the ENS group for their research, each component will need to be completed. These requirements differ from the requirements defined during the initial research phase, which can be found in Appendix B.4. The requirements in the research phase were defined for the initial prototype, which mainly focused on communications between the server and the Android application. Later on more components and features were included in this project and thus the requirements were redefined for the final product. A list of the requirements defined for the final product can be found in Appendix D.

3

ARCHITECTURE DESIGN

In this chapter, the architecture of the system will be described and elaborated on. Firstly, a general overview will be given. Then, the responsibilities of the different components will be clarified.

3.1. DESIGN GOALS

Two design goals that were the most relevant are maintainability and scalability. These two design goals were kept in mind during the entire design process of the system and will be used to evaluate whether the product is successful or not. Explanation regarding these design goals can be found in Appendix B.3.

3.2. SUBSYSTEM DECOMPOSITION

The final product has several components: Server, Android application, microcontroller, RFID reader, reader interface for RFID reader, database, and cache. The system is designed such that messages from the Android application can be transmitted to the RFID reader through the microcontroller, which encodes the message received from the Android application. This message is a command to modify an EPC, and After the RFID reader decodes this message, it can be sent to the server together with an EPC to indicate which EPC will need to be updated. Finally, the server has access to the database with all information on the EPCs. The server will execute the command and send feedback to the Android application. An overview of the system can be seen in Figure 3.1.

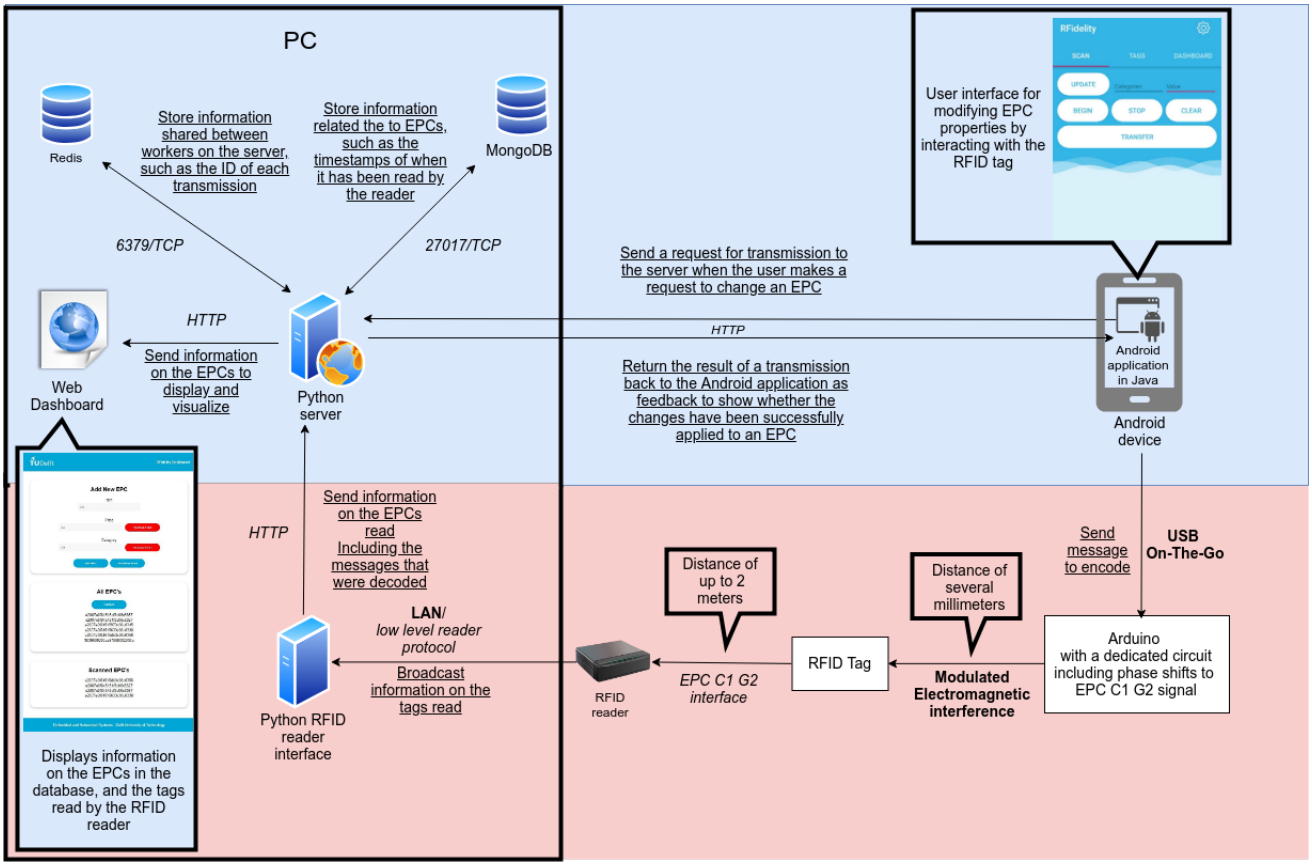


Figure 3.1: Overview of the final product with all subcomponents. The components in the blue area are developed during this project. The components in the red area were developed outside of this project, connecting and integrating these components into the complete system was part of this project.

SERVER

The server is written in Python with Flask as the web framework. It is responsible for communication between several components. It is the only component that is directly connected to both the cache and the database. Communication with the server is mainly done with HTTP requests, documentation of the API can be found on the repository of the server.

The choice for Python was made as this was a project requirement (see Appendix C), due to the readability of Python. Another reason why Python was used instead of for example Node.js was that the code that was already being used in components of the existing research project was already written in Python and had to be adjusted. Thus using Python would allow for an easier transition from their current product to this product. One of the issues with choosing Python over Node.js was performance. As Node.js is generally faster than Python and Python does not work well with multithreading. This issue is solved using a combination of Flask with Gunicorn.

Flask [1] is used for the web framework, as it is more suited for small projects compared to other frameworks like Django [2] and Pyramid [3] [4] as Flask is a microframework, while both Django and Pyramid are full-stack frameworks. This means that Flask only includes a minimal amount of modules, including modules that are required in this project, such as request and response handlers. While full-stack frameworks provide more modules, this can hinder the development process when these built-in modules have to be avoided when other tools are chosen for the same task. However, Flask by itself is not scalable and not intended for production, as it can only handle one request at a time. For this reason, another framework, Tornado, was considered. However, Tornado is less well known and thus has less information that can be found when is-

[1] http://flask.pocoo.org/
[2] https://www.djangoproject.com/
[3] https://trypyramid.com/

sues occur, this will be undesirable for developing and maintaining the product. In the end, the decision was made to use Flask with Gunicorn [4] for deployment, this will solve the scalability issue of Flask, while keeping the benefits of Flask. Furthermore, Gunicorn also resolves the issue regarding Python and multithreading. Gunicorn can spawn multiple workers, which each are Python processes. The requests to the server can be dispatched to these workers and each worker will be able to process their request.

ANDROID APPLICATION

The Android application is written in Java. The user interface of the app is written in XML. It is used as the interface for the user who wants to interact with the RFID tags, and it allows users to edit information of an EPC associated with this tag by entering a message that will be piggybacked on the information received by the RFID reader from the RFID tag. Feedback from the server will be shown in the application after the server has processed the message. Figure 3.2 shows the Android application designed in this project.

The choice for Java over Kotlin for the programming language was made because the team was more familiar with Java and the advantages that Kotlin has over Java did not outweigh this. The advantages found during the research phase can be found in Section B.6.3.

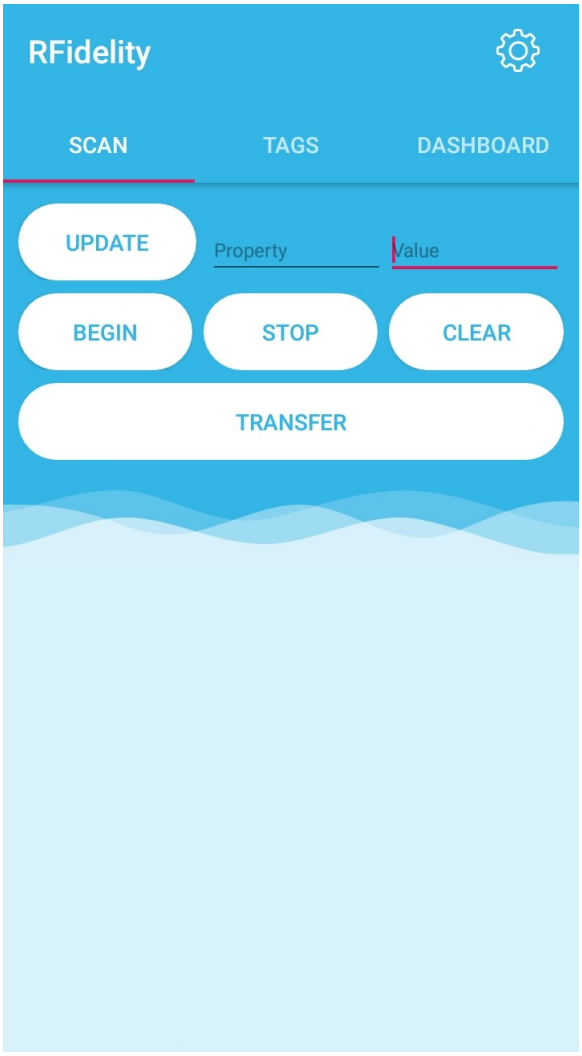


Figure 3.2: Screenshot of the Android application. Properties and values are used to modify the properties of the EPC. Begin and stop are used to connect and disconnect with the microcontroller. The transfer button will initiate the transmission using the property and value filled in.

[4] https://gunicorn.org/

Figure 3.5: Picture of the RFID reader connected to a laptop running the reader interface

MICROCONTROLLER

The microcontroller is responsible for changing the phases of the RFID tag such that the RFID reader will receive the information on this tag together with the additional message that was encoded on the Android application. An *Arduino Nano* was used during the development of this product. This Arduino was connected to a coil. The setup of this Arduino was provided to the team. During the early development a standalone Arduino was used, this was expanded to the setup currently in use by the ENS group in their research. The final setup of the Arduino can be seen in Figure 3.3 and Figure 3.4.

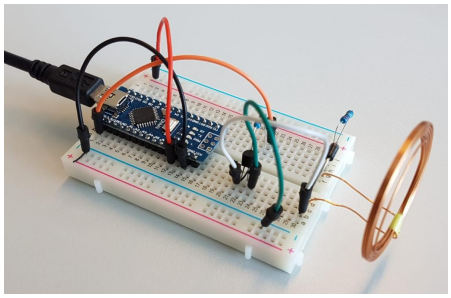


Figure 3.3: Picture of the microcontroller with its dedicated circuit

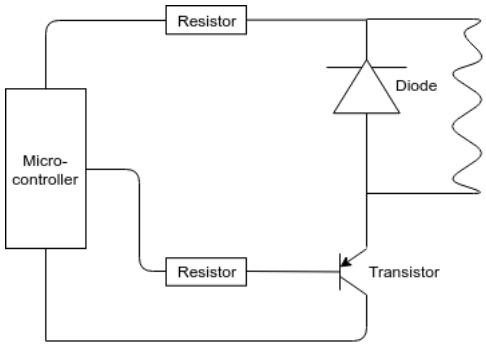


Figure 3.4: Diagram of the dedicated circuit for the microcontroller

RFID READER

The reader is used to power the tags in its range, the tags that have been read will be relayed to an interface that allows for communication with the other components. The project used the *Impinj IPJ-R1000-USA1M3* during its development. The setup for the reader can be seen in 3.5.

READER INTERFACE

The reader interface is written in Python, it obtains information on the tags that have been read from the RFID reader and represents the information in a standardized structure by using Sllurp [5], which is a Python

[5] https://github.com/ransford/sllurp

implementation of the low-level reader protocol. The interface will be responsible for relaying information on the tags that have been read to the server.

The interface is in Python because of the existing code for interacting with the RFID reader. The initial code was used as the basis for the reader interface in this project.

DATABASE

The database used is MongoDB, it is used to store information on the tags and products. Each tag will be a separate item in the database (see Fig 3.6). Each item will have the EPC as a unique ID of the tag associated with the item. Other than the EPC it will also have the field *properties* to store all additional information related to the product. Outside of *properties*, three more fields were included as a list of values in the database, *time*, *seen_time*, *count*, and *RSSI*. These fields were added separately because *properties* are used to store information related to the product that the tag is attached to, while these additional fields store information that is related to the process of reading the tags.

The field *time* keeps track of all timestamps when this EPC has been modified in the database. *Seen_time* is used to show all timestamps when this EPC has been read by the RFID reader. *Count* keeps track of how many times the EPC has been read within one cycle of the RFID reader, this is not equal to the total amount of times the EPC has been seen. *RSSI* are the received signal strength indicators (RSSI) of each tag that has been read by the RFID reader, this is to indicate how strong the signal was when the tag was read. *Timestamps of reads*, *count*, and *RSSIs* have the same amount of items in their lists. Using the indices the three lists can be combined to obtain the information on one EPC read during a cycle of the RFID reader.

Figure 3.6: Example of an EPC in the database

CACHE

The cache is a Redis database, it is used to store information that needs to be accessed between different requests to the server, such as information on pending requests from clients who have sent a request to modify some information in the database. Another use case for the cache is to store the EPCs that have been read in the last cycle of the RFID reader. In both cases the information will need to be shared between different requests and the information will only need to be available for a limited duration. The cache will not keep its state between different sessions, unlike the database.

The need for shared memory between requests was not known during the research phase. During the research phase the decision to use Python with Flask and Gunicorn was made, Gunicorn uses multiprocessing to allow for scalability. This means that each separate worker in Gunicorn will not share its information with

the other workers, thus to allow for the functionality required the decision was made to use a Redis database as a cache.

Another option that was considered was using the *preload* function of Gunicorn, this would allow some memory to be used as shared memory before creating new workers. However, this is mainly used for static data, while the data that needs to be stored in the use cases of this project are dynamic. Furthermore, Redis provides builtin functionality for expiring items in the database. This way the server will not have to handle this task of checking for expired items. Because Redis stores all key-value pairs in memory, it is relatively fast for simple operations compared to other databases. Especially for the task of storing all EPCs that have been read in the last cycle.

3.3. DASHBOARD

To make the information stored in the database more accessible to the user, a dashboard was created. This dashboard is a web page that allows the user to add new EPCs, show all EPCs in the database, and show all EPCs read by the RFID reader. The dashboard was made to improve the usability of the system, by allowing the state of EPCs to be inspected and manipulated without requiring knowledge on how the server and databases work.

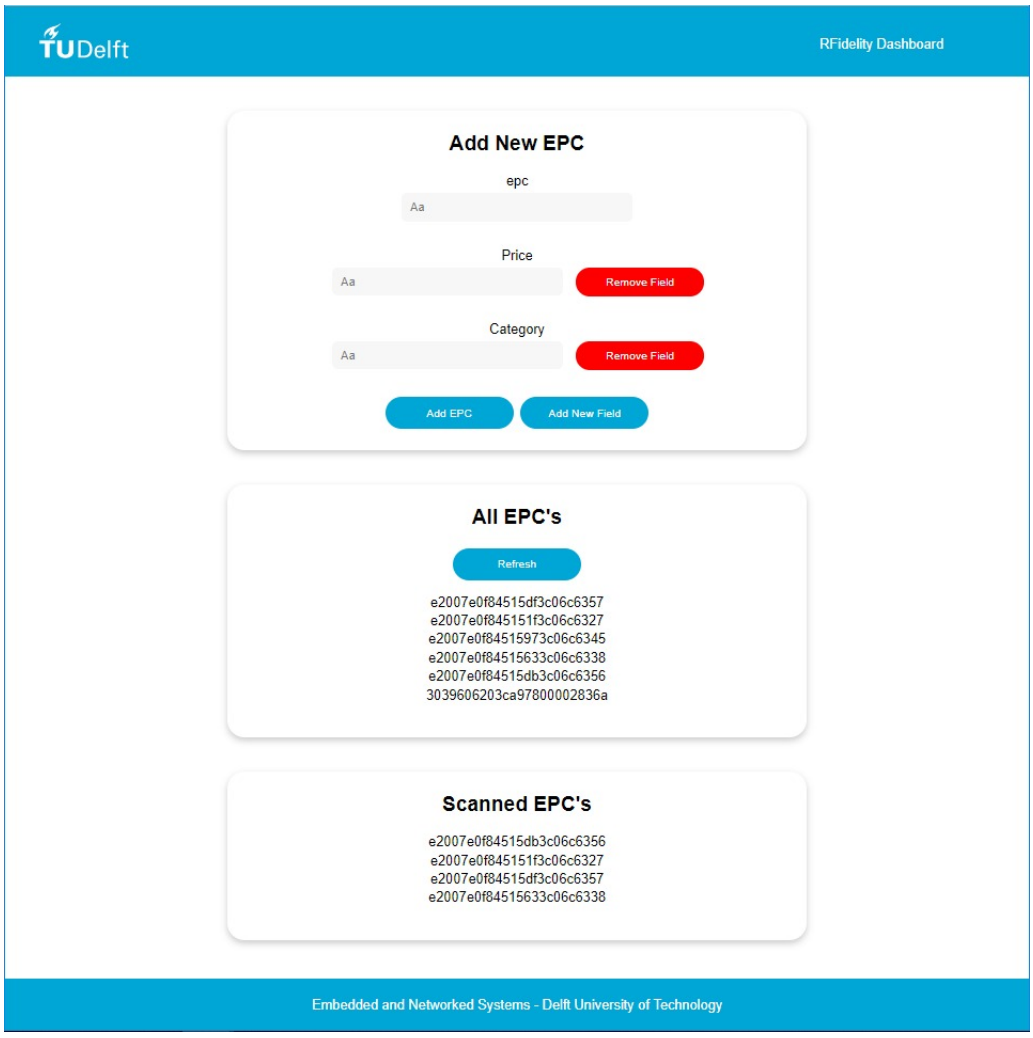


Figure 3.7: Screenshot of the dashboard

3.4. CACHING

Redis stores all entries as key-value pairs, where the key is a string. Different types of items can be distinguished by using prefixes, Redis can match the keys with patterns to retrieve information. In this project, the keys that are stored in the cache are EPC, request, and read tags.

EPC

EPC was the first key added to the cache, the purpose of this key is to indicate to the server that this EPC has been scanned and the server should expect confirmation soon. An error has occurred if no confirmation is given before the key expires. These EPC keys use the following prefix:

```
__epc__
```

REQUEST

Request keys are used to indicate that a transmission has been started and the server should expect the RFID reader to send a confirmation. In a similar way to the EPC, an error has occurred if the server does not receive a confirmation before the key expires. The request keys use the following prefix:

```
__host__
```

Furthermore, for a request, another type of key is stored, this key maps a host to a port and is used to send the feedback back to the client who made the request.

MESSAGE ID

When creating a request on the Android device, the client will need to request a unique ID from the server. Each ID is an integer used to indicate which message and client are associated with this ID, this information is stored in the cache. The number of unique IDs available can be configured on the server. Each ID will make use of two keys, one with as key the ID and as value the message associated with this ID. The other key will use a prefix in the form of:

```
__id__
```

This key will map to a host and is used to be able to return feedback to the client after the request has been processed.

READ TAGS

The information on the tags that have been read is available in the database, as each EPC stores all timestamps of when it was read. However, to find all EPCs that have been read in the last cycle would require aggregation of this information after retrieving all EPCs from the database. To reduce the load required to find this information, an alternative solution was used. The EPCs of the tags read in the last second are stored in the cache and will expire after 1 second. This is to provide a way of inspecting which tags are being read by the reader. The prefix used for these keys is as follows:

```
__lastseen__
```

3.5. API

The server has a RESTful API to allow for communication with it. This choice was made to allow for scalability as well as extensibility. Scalability is because a RESTful API is stateless - it does not need to remember a client session. This means that each client will be treated independently, and the server could be distributed if required. Extensibility is because more routes can be easily added later on, using a well-defined namespace will ensure that existing routes can be easily found and new routes can be easily added. The API in this project contains two branches, routes for the database and routes for the cache. The routes for the database use `db`. The routes for the cache use `cache`. This separation was made to ensure that routes for similar functionalities, like adding items either to the cache or database, are easily distinguishable.

To make sure that the API can be easily used and maintained later on, API documentation was made. In this documentation each route is described, as well as the parameters that the routes could have. To make it more clear, an example of both the input and output is given as well.

3.6. COMMUNICATION

To create the product according to the description, the required components were analyzed and separated where necessary. By separating the components, each component would have a clear role in the system. This way the product will be easier to maintain later on. One of the main focuses of the project was in developing these components and allowing them to communicate with each other. Each component has a different way of communicating with other components.

The Android application communicates with two other components, namely the server and the microcontroller. Communication with the server is done in two ways, the first being through the API of the server. However, this only allows for requests being started from the Android application. In the case that the server needs to respond to the client. Later on, a possibility for a socket connection was built. This allows the server to make a socket connection with the Android application to return feedback. Communication with the microcontroller is done through a serial connection, using a USB cable. This means that the Android device will need to be connected to the microcontroller physically for the system to work.

The microcontroller is connected to the Android device and gets messages from the Android application through a serial connection. The microcontroller is also responsible for interacting with RFID tags using modulated electromagnetic interference.

The RFID reader reads information from the tags and communicates these with the reader interface, this is done using a TCP connection. In the setup used during the development, a local network was created using a router. Both the RFID reader and the machine running the reader interface would be connected to this router. The EPCs that have been read by the RFID reader would be broadcasted on this network.

The reader interface would use the API of the server to communicate with it, sending information on the EPCs that have been read to the server.

Lastly, the server has access to both the database and the cache. Which allows it to modify the information stored in them.

3.7. UNIQUE ID FOR MESSAGES

As earlier discussed during the development, the messages containing the command to be executed on the EPC were encoded in the information received by the RFID reader from the RFID tags. However, during the testing of the encoding and decoding system that was provided to the team, it was noticed that the results were rather flaky, even for transmitting 8 bits. Sending the messages containing the command would be infeasible using this method. Thus a different approach was designed, in this new approach, the Android device would send the command to the server directly and encode a unique ID obtained from the server in the messages received by the RFID reader. These unique IDs can be a lot shorter depending on the number of concurrent requests required. Which will be much more feasible compared to the initial design that was proposed.

3.8. ERROR DETECTION AND CORRECTION

As the transmission of the encoded messages received by the RFID reader is often prone to errors, measures to detect and correct some of these errors have been included. Hamming code is used to correct up to 1-bit errors and detect up to 2-bit errors. At first, (7, 4) Hamming code was implemented, where the total message contains 7 bits out of which 4 bits are reserved for data. This implementation would allow for the detection and correction of a single bit, it would not be able to distinguish a 1-bit error from a 2-bit error. This could result in an attempt to perform error correction of 1-bit, while 2-bit errors occurred. To improve this implementation an additional bit was added to allow for distinguishing 1-bit errors and 2-bit errors. The implementation would still be unable to correct 2-bit errors, but it will not attempt to correct them as if they were a 1-bit error.

4

FINAL PRODUCT

In this chapter, the different functionalities of the product will be outlined. Storing and updating EPC information, encoding and decoding information, message transmission, read tags and RSSI charts will be covered.

4.1. STORING EPC INFORMATION

For the product to function, information on EPC will be stored beforehand in the database connected to the server. Several ways to modify EPC information in the database are included, using the Dashboard, making direct requests with the API, importing JSON files using Python.

Using the dashboard, the user can add new entries to the database or edit existing ones. This is done by filling in the forms as key-value pairs, one such pair is the EPC itself, this pair is required. Other pairs can be manually added by the user, depending on the information required for the EPC. Once all information has been filled in, a request will be sent to the server, if the EPC does not exist it will add a new entry with all key-value pairs other than the EPC as its properties. If the EPC already exists, the EPC will be replaced with the new entry, which means that any fields that are not included in the new version will be removed.

Requests can also be directly made with the API, by using the `/upsert/<epc>` route. This route expects a JSON object to be included in the request, this JSON object will store all information on the EPC.

On the server-side, EPC can be added using a python module. This will allow for adding multiple EPC at once. Each EPC will be stored as a separate JSON file, the script will look for these JSON files in the specified folder and import each one to the database. This is built to allow data to be exported from MongoDB to be imported back to the database through Python, which is also used for setting up test databases.

4.2. UPDATING INFORMATION

To update the information of an item with an RFID tag, a user will start the process on an Android device with the application installed on it.

To change the property of the item, the user will first request a unique ID from the server. This ID is used to distinguish different messages that were sent. After getting a unique ID from the server, the device will send another request to the server. This time the request will contain the unique ID obtained from the server together with a message that encodes a command to be performed on an EPC. The server should expect a confirmation from the RFID reader within a certain duration. If no confirmation is made during this duration, the server will inform the Android device to retry.

Another message required to start the process is a message to the Arduino, this message will only contain the unique ID obtained from the server. This unique ID will be encoded in the information received by the RFID reader.

The RFID reader can then decode the message that was sent along with the information from the RFID tag. The RFID reader will send a request to the server indicating that an EPC was found with a unique identifier attached to it. The server will look this ID up in the cache and upon successfully finding it, it will execute the command that was encoded in the message on the EPC. After executing the process the server will send a success message as feedback to the Android device. In the case that the request timed out, the server will notify the client of this and the client can decide to restart the whole process.

4.3. ENCODING AND DECODING INFORMATION

To modify information on an EPC, the user will need to enter a message on the Android device. This message will contain a command to be executed on the EPC and will be sent to the server. This message will contain information on what operation needs to be performed, this can be either 'u' or 'r', where 'u' is for update and 'r' is for remove. Update operations require two additional parameters, a property, and a value. The property is used to indicate which property of the EPC will need to be changed and the value is the new value for the corresponding property. An example of an encoded message for updating is:

```
u_price_50
```

The '*u*' is used to indicate that this is an update operation, '*price*' is the property that needs to be updated and '*50*' is the new value for the property.

Remove operations only require one parameter, which is the property that needs to be removed. An example of an encoded message for removing is:

```
r_price
```

The '*r*' is used to indicate that this is a remove operation, '*price*' is the property that needs to be removed.

Another message that will be encoded during this process is the unique ID obtained from the server, this unique ID will be a number used to indicate which message was associated with the EPC. The encoded unique ID will be sent to the Arduino, where it will be used to cause phase shifts in the signals received by the RFID reader. The RFID reader will use these phase shifts to decode the attached ID to allow for updating information.

The initial approach for encoding and decoding messages would encode the command with the changes that need to take place. However, this approach was found to be inefficient and infeasible with the current system. This is because the RFID reader would struggle with receiving the correct encoded information. Each extra bit of information will heavily impact the success rate of the transmission. Thus to reduce the amount of information transferred through this channel, this change was made to the new approach.

4.4. RETRANSMISSION/CONFIRMING MESSAGE

If the server has processed the confirmation from the RFID reader, it will send feedback to the client. This feedback echoes the message that the reader has received back to the client. This is used to check whether the RFID reader has received the correct message. The client will need to manually check whether the transmission was performed correctly upon receiving the feedback. The reason why this is not automated is that transfer requires the microcontroller to be placed near the tag. Retransmission will fail if the microcontroller is not near the tag, thus automating this process would limit the range of movement for the client while waiting for a response from the server.

If the server does not get any confirmation from the RFID reader within the expiration time of the request, it will send a message to the client indicating that the transmission has failed. In this case, the user can start the broadcast again by repeating the process, in a similar way to retransmission when the RFID reader received an incorrect message.

4.5. READ TAGS

The reader interface will relay information on the tags that it has read to the server, this information will, in turn, be stored in the database. To make this information more accessible, the last read tags will be included on the dashboard. This list gets updated every second and the names of all EPCs that were read in the last second will be displayed.

4.6. RSSI CHART

One of the important fields of an EPC stored in the database is the RSSI. This shows the strength of the signal received by the RFID reader. To make this property easier to access, a chart was created on the dashboard to inspect the RSSI of each EPC over time. This chart allows the user to select the EPC that needs to be inspected using a drop-down list. Each EPC will have the RSSI on the vertical axis and the time on the horizontal axis. An example can be seen in Figure 4.1.

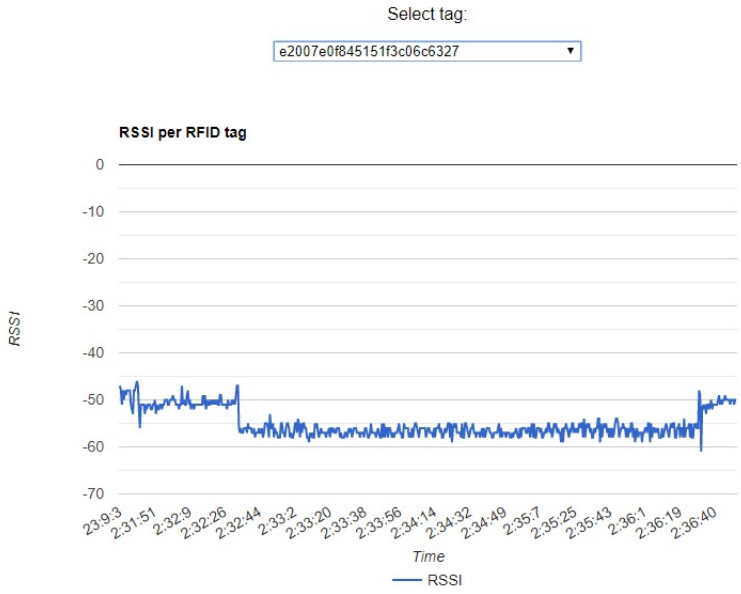


Figure 4.1: Line chart showing the RSSI of an EPC overtime

This chart was one of the features included as a tool to help with the research of the ENS group. At the start of the project, a similar chart that was used was shown. However, that chart was located on the RFID reader and would only show the information about the current session. The RSSI chart on the dashboard obtains its information from the database, which means that the user can access the results from other sessions as well, as long as they are stored in the database.

5

TESTING

The activity of checking whether the software is working as it is intended to work is an essential part of the software development process. A working and bug-free software product that is validated using testing, as set out in the design goals, is expected. Many types of testing were applied during this project. In this chapter, the testing strategies that were used will be presented. In Section 5.1, the application of functional testing in the final product will be described, whereas Section 5.2 will introduce the non-functional testing,

5.1. FUNCTIONAL TESTING

This section will describe the different types of functional testing used during this project. The functional testing performed in this project can be separated into three categories, namely unit testing, integration testing, UI testing and end-to-end testing.

5.1.1. UNIT TESTS

For the server, which is written in Python, the `Pytest`[1] framework was used. Pytest was chosen as the testing framework because it supports tools that make testing easier. One such tool that was used during this project was fixtures in Pytest, this allows behaviour for setting up and tearing down tests to be defined and used globally. These fixtures were used to populate the databases before executing a test. On the Android application the `JUnit`[2] framework was used for unit testing. JUnit is built-in to Android studio and allows for quickly executing the unit tests. The unit tests check that each function on the server shows correct behavior when called upon. These tests were run automatically using Gitlab CI, by automatically executing these tests failing tests would be fixed before getting merged back into the master branch. Thus ensuring that the code on the master branch works as intended. (see Section 7.3).

Branch coverage and line coverage were used to indicate whether the code was tested sufficiently. Parts of the code were excluded, as these were not part of unit testing. These parts include the API routes, they are instead covered by integration testing. The coverage can with the routes be seen in Figure 5.1, the coverage reaches up to 77%. The routes mainly have statements that are missing in the coverage. PyTest does not register calls to these routes as they are running on the server that needs to be up for these tests. Excluding the routes will increase the code coverage to 96% as can be seen in Figure 5.2. Some branches are still uncovered, this is because these branches are used to ensure that the CI has the correct settings when ran. These will not be executed when running locally and thus are not covered. For example, the URIs to connect with both the database and the cache is defined (see Figure 5.3).

[1] https://pytest.org
[2] https://junit.org/junit4/

```
---------- coverage: platform linux, python 3.7.3-final-0 ----------
Name                          Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------------------
app/__init__.py                   0      0      0      0   100%
app/config.py                    15      0      4      2    89%   6->10, 11->14
app/db/mongo_util.py             36      1      6      0    98%   77
app/db/redis_util.py            108      0     28      2    99%   198->exit, 214->exit
app/rfidelity.py                 25      3      4      2    83%   11, 43, 47, 10->11, 46->47
app/routes/cache_api.py          61     38     18      0    29%   14-18, 24-28, 36-41, 48, 54, 60-61, 69-82, 89-100, 106, 111
app/routes/charts_api.py          5      1      0      0    80%   11
app/routes/dashboard_api.py       5      1      0      0    80%   11
app/routes/db_api.py             36     19      6      0    40%   14, 20, 25-26, 32, 38-52, 59-60
app/util/epc_message.py          17      1      6      1    91%   23, 10->23
app/util/response_socket.py      26      0      4      0   100%
-------------------------------------------------------------------------
TOTAL                           334     64     76      7    77%
```

Figure 5.1: Code coverage server with routes

```
----------- coverage: platform linux, python 3.7.3-final-0 -----------
Name                          Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------------------
app/__init__.py                   0      0      0      0   100%
app/config.py                    15      0      4      2    89%   6->10, 11->14
app/db/mongo_util.py             36      1      6      0    98%   77
app/db/redis_util.py            108      0     28      2    99%   198->exit, 214->exit
app/rfidelity.py                 25      3      4      2    83%   11, 43, 47, 10->11, 46->47
app/util/epc_message.py          17      1      6      1    91%   23, 10->23
app/util/response_socket.py      26      0      4      0   100%
-------------------------------------------------------------------------
TOTAL                           227      5     52      7    96%
```

Figure 5.2: Code coverage server excluding routes

```
class BaseConfig(object):
    REDIS_URI = os.environ.get('REDIS_URI')
    if REDIS_URI is None:
        REDIS_URI = 'redis://localhost:6379/rfidelity'
    MONGO_URI = os.environ.get('MONGO_URI')
    if MONGO_URI is None:
        MONGO_URI = 'mongodb://localhost:27017/rfidelity'
    PORT = 5000
```

Figure 5.3: Snippet of branches that are not covered by the tests

5.1.2. INTEGRATION TESTING

For the server, integration testing was used to verify that the API on the server shows the intended behaviour. Two different strategies were used to test these routes. The first one tests each route separately, verifying that the logic performed when calling this route is correct. This is done by using a setup phase before the tests are executed. This allows the state of the system to be defined for each test and the expected result will not be reliant on the previous tests executed. The second strategy used was to chain these routes to mimic a use case of a user. This is done by defining several steps in the test, where each step performs a specific action. This strategy allows the combination of routes to be tested, to verify the behaviour of the system.

When designing the tests for the routes, multiple tests were created for each route. This is to ensure that not only good weather behaviour was tested, but also possible incorrect inputs. For example, when adding items to the database, tests were created to check for the cases where an item would be added, as well as the cases where a request would be denied due to incorrect or missing parameters of the request.

In both cases, the server will need to be started on the local host. These tests are also included in the GitLab CI, to ensure that changes to the code base will not adversely change the behaviour of the system.

Components like the Arduino were difficult to automate for testing, as it requires a physical device to be available and be connected to an Android device. To make sure that the connection with the Arduino would still function as intended, manual testing was performed whenever deemed required. These tests were only required when making changes to the Android application, as the other components would not directly interact with the Arduino. Another component that was difficult to test was the RFID reader, as the setup of the reader was placed in the office and was not available outside this office.

5.1.3. UI TESTING

Some of the code in the Android application is logic that can be tested using unit tests. However, the application also contains code related to the user interface (UI). To be able to test the UI, the `Espresso`[3] testing framework was used. Espresso tests can be executed on a physical phone or emulator, which is very convenient for debugging as one can see what is happening during the runtime of the tests. Furthermore, Crashlytics was added to the Android app. Crashlytics will report in real time crashes coming from devices running the app. Once a crash occurs, it is effortless for the developers to trace back the error from the incoming crash report coming from Crashlytics. The type of device on which the crash occurred is reported in the crash report. This is beneficial when an error is related to the user interface as it is tough for the developers to test the app on all devices running Android.

5.1.4. END-TO-END TESTING

To verify that the system as a whole behaves as expected, end-to-end testing was performed. These tests were designed to test possible use cases of the system for when it will be used in production. Two distinct test scenarios were defined, one for testing the process of updating an EPC using the Android device. The other test scenario was used to test whether the RFID tags are read and displayed on the dashboard properly.

To test the process of modifying an EPC, all components would be given a fresh start by restarting the server and RFID reader and by clearing the cache and the database. This is done to ensure that the current state of these components would not influence the tests, as the tests are run on the same machines used for development. After setting everything up, the process would be started by initiating it on the Android device, the first test with this scenario is to check whether the Android device gets the correct feedback from the server if the RFID tag has been read. The next test is to remove the RFID tags from the range of the RFID reader and check whether the request will timeout properly. For both tests the encoding and decoding of message through the microcontroller and RFID reader are not used, this is because this feature is still unstable and could affect the results of the tests.

The other scenario that was tested was to ensure that the communication of the RFID reader with the server was working properly. This was done by placing multiple RFID tags in the range of the RFID reader and verifying that the EPC of these RFID tags are displaying on the dashboard. Multiple configurations are used for this test, one with all 6 tags in the range of the RFID reader, one with only a single tag in the range of the reader, and lastly one with no tags in range. These tests were done in succession to ensure that the dashboard was updating its information properly.

5.2. NON-FUNCTIONAL TESTING

5.2.1. LOAD TESTING

Load testing was performed to ensure that the server would be able to handle multiple concurrent requests. This was done using the Python framework *Locust* [4], the framework allows user behaviour to be defined in code. All tests were run on an *HP ZBook Studio g5 x360* with an Intel Core i7-8750H (six cores) processor and 16GB RAM. The number of users can then be set during execution. To test the load on the server, 4000 users were used as this was the maximum number of users that the laptop could handle, changing the configuration of the machine where the server will be deployed will resolve this issue. With 4000 users, the requests per second (RPS) could go up to 500. In this test, each request would be the same request. The request would be to retrieve all information from the database, where the database contained 55 EPCs with a total size of roughly 7KB per request. This test was run using Gunicorn with 4 workers and 1000 connections per worker. The same test was run using only Flask, in this case, the server would get up to 40% error rate with the requests. Furthermore, it also had a much lower RPS and higher average response time. The results from both tests can be seen in Figure 5.4 and Figure 5.5.

# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
69443	3	150	2046	3.370046615600586	139159.40380096436	7672	431.4

Figure 5.4: Load test Gunicorn with 4000 users and 55 items in the database

[3] https://developer.android.com/training/testing/espresso
[4] https://locust.io

# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
51945	1726	500	5181	3.6318302154541016	157350.94547271729	7417	283.5

Figure 5.5: Load test Flask only with 4000 users and 55 items in the database

Another test was run to check for the performance when the database stores a large amount of data. For these tests, the database was populated with 10000 items, with a total size of nearly 2MB. In this scenario, the test with flask application had roughly 2 requests per second, while the test with Gunicorn had roughly 6 requests per second. Both scenarios were run two times to ensure that the results were consistent, both the server and the load testing tool would be restarted after each run. The tests would be run for roughly 400 requests. Both tests with Flask showed that the responses took nearly 3 times as long as the responses of the Gunicorn tests. The Flask tests had a median response time of 71 seconds and 90 seconds, while the Gunicorn tests had a median response time of 25 seconds and 26 seconds. It was figured out that the database access was the bottleneck of the system. This is because, during these tests, the server would still be responsive to requests that do not require access to the database. Furthermore, the Flask tests would result in failures as the test went on. This is due to the requests being timed out by the server. The results of these tests can be found in Appendix F.

6

EVALUATION

In this chapter, the final product will be evaluated with regard to different aspects. Section 6.1 will evaluate the final product in terms of the requirements. Section 6.2 will assess to what extent the design goals have been kept in mind. Lastly, Section 6.3 will evaluate the product as a whole.

6.1. REQUIREMENTS EVALUATION

The requirements for the final product have changed throughout the project, as more components were included after the research phase, as well as a more complete picture of the whole system, was drawn. Furthermore, the requirements set during the research phase were tailored for the initial prototype. It was indicated that the focus was to be put on the initial prototype first and slowly building up on top of that.

All *must have* requirements defined in B.4 have been implemented. These requirements focused on the communication aspects of each component, which is essential for the product, despite the inclusion of new components. On top of this the requirements for communications between the new components, such as the RFID reader, were added.

From the *should have* requirements, most have been included. The requirement to allow the user to define the syntax of an EPC has not been implemented. This is because the need for this was removed.
In the initial research phase, it was found that EPC contains information that can be used to uniquely identify a tag, this EPC can include information such as the origin of the tag. However, it was indicated later that additional information on the tag should be stored in the database, and will be added manually beforehand to the database. Thus removing the need for this information to be stored in the tag itself. This allows for modifications of data without changing the information on the tag itself. Thus it was not necessary to allow for specifying the syntax of an incoming EPC, as the EPC is only used as a unique id for each tag.

6.2. DESIGN GOALS EVALUATION

During the initial research phase, two design goals were identified as the most important ones for this project. These were maintainability and scalability.

6.2.1. MAINTAINABILITY

As mentioned in B.3.1 maintainability is an important aspect of this project. This is because the product will be used by many people and will be further developed to continue research. Several sub-goals were identified in the research phase, namely code quality, testability, and documentation.

CODE QUALITY

Code quality was ensured by the use of GitLab's merge request feature before including changes to the code base to the existing product. This made sure that at least one other group member had to approve of the changes, while this sometimes ended up not being sufficient as each group member had a different view on

maintainable code. The code has also been submitted to the Software Improvement Group (SIG), SIG also does code reviews to rate the codebase based on their metrics. Their focus is on checking for maintainability of the code. SIG has given 4.6 out of 5 stars on the first submission (see Appendix E), with some minor remarks regarding unit size. Following their feedback, the two functions that have been given as an example by SIG have been refactored and split up. This way each method does not have too many responsibilities. During development tools like linters were used to ensure that the code style is consistent and to detect possible code with unintended behaviour. For the Python code, the linter Pep8 was used and for the Android studio the linter Android lint was used.

Another tool to check for the code quality was Radon, a Python tool that computes various code metrics. Some of the metrics that were checked are cyclomatic complexity and the maintainability index. For the cyclomatic complexity, it was found that all functions had the highest rank (where a higher rank is better). The cyclomatic complexity of these functions was between 1 and 5. A single function was found to have a cyclomatic complexity between 6 and 10. This is because this a function is a route that deals with a tag that has been read, this function checks that all the required parameters are present and thus increases the cyclomatic complexity. For the maintainability index, Radon showed that all modules were of very high maintainability, as each module scored the highest rank.

A possible issue regarding maintainability could be found with using Python on the server and RFID reader, as Python uses dynamic typing. It is debatable whether languages that use static typing are more maintainable compared to languages that use dynamic typing, however it was found that for several programming tasks static typing does help [12]. Python provides the possibility to use type hints, this allows developers to indicate which type should be used, without enforcing it. This will reduce the amount of effort required when a new developer explores the existing codebase.

TESTABILITY

Most components like the server and the Android application use automated testing to ensure that the system shows the correct behaviour. However, components like the Arduino and the RFID reader are not tested automatically. These components are tested manually as part of the whole system using end-to-end testing. The reason for this is that these components require the physical device to be attached to the system for it to work. Automated testing is done using continuous integration (CI) on GitLab. Each time that the code has changed in the repository, the build will be automatically tested to ensure that the changes did not break the existing behaviour.

The communication between each component is not tested automatically, the communication is instead mocked during testing. This means that changes to the API on the server will not be caught automatically by the other components that use this API. The tests on the other components would still pass, as the result from the API calls are mocked.

DOCUMENTATION

To increase the maintainability of the product, documentation was made for the API. This allows users to be able to interact with the server without knowing about the inner workings of the server. This documentation is included in the repository of the server. Other than the API documentation, a startup guide has been included for the users of the product. This guide lists all requirements for the product to work on their machine. As well as the steps required to set everything up.

6.2.2. SCALABILITY

To mimic an RFID system in a retail environment, the system would need to be able to handle concurrent requests from multiple clients. The bottleneck in the design of this system would be the server, to solve this *Gunicorn* is used outside of development. *Gunicorn* takes care of the logic required to spawn multiple workers, each worker is an instance of the Python application. The requests to the server can then be dispatched to these workers, where each request will be processed by a single worker. To improve this even further, Gunicorn allows for different types of workers to be used. One such a worker is an Asynchronous worker, these workers can handle multiple requests simultaneously, compared to a standard synchronous worker that can

only process a single request at a time.

Another possible bottleneck would be the requests from the RFID reader. Each tag that has been read by the RFID reader will be sent to the server to be stored in the database as well as in the cache. If the number of tags read by the RFID reader increase, it could cause too much traffic for the server. For this to happen the RFID reader would need to read more than the limits found during load testing (see Section 5.2.1). The RFID reader used during development was able to read up to 800 tags per second with 6 tags in the range of the reader. These reads, however, are aggregated by each tag. This means that in the case of 6 tags being present, only 6 requests would be sent to the server. However, if a sufficient amount of tags would be placed in the range of the reader, it would be possible to exceed the number of requests that the server can handle.

6.3. PRODUCT EVALUATION

During development the setup used when testing the product was a static environment, which means that everything was placed on a fixed location. The Android device would be physically connected to the microcontroller, but the microcontroller with the circuit attached to it would be located directly next to the RFID tag. One of the tests performed was to attempt to move around with the Android device and the microcontroller to mimic a user walking around with a handheld RFID reader. The RFID tag was placed on a location within the range of the RFID reader, where the distance to the RFID reader is similar to the distance used during development. The user would approach this tag with the microcontroller and attempt to transmit a message. Several attempts were made, however, the RFID reader would pick up too much noise during this test, this is possibly due to the movements of the hand when holding the device. Even during development, the microcontroller and RFID tag will need to be positioned perfectly for the optimal result.

Another evaluation that was performed was regarding the time it takes to complete a cycle of modifying an EPC. The setup of this test was set up to check for both the time required for the communications only, as well as the time required including the decoding and encoding of the messages received by the RFID reader. Furthermore, another distinction that was made was with regards to the state of the database. Two cases were defined, one where the database only contained a few test items and another case where the database contained 10000 items. The results from the 4 different test cases run 100 times each can be found in Figure 6.1.

	6 items in the database with encoding/decoding	6 items in the database without encoding/decoding	10000 items in the database with encoding/decoding	10000 items in the database without encoding/decoding
Average (ms)	2037.93	696.33	1958.61	771.84
Median(ms)	2051	719	1948	831.5
Standard deviation (ms)	103.50	285.47	78.43	275.66
Confidence interval (ms, $\alpha = 0.05$)	20.28	55.95	15.37	54.03

Figure 6.1: Results from testing the delay of the system with different configurations, each test was run 100 times.

7

PROCESS

In this chapter, the process of the project will be described. The process is crucial to ensure the team remains effective and on schedule. In particular, the used development methodology and workflow will be described and evaluated.

7.1. DEVELOPMENT METHODOLOGY

During the start of the project, the team decided to use Scrum. It makes a project manageable and team communication improves and customers see incremental deliveries [25]. Using Scrum, everybody has his issues and can work on that certain part of a project independently. At the end of every sprint, the team would meet with the product owner to discuss the current state of the project, as well as the future development of it. During these meetings adjustments were often made to the design and new features or components were requested. However, later on in the project the meetings were sometimes not held at the end of the sprint. Because the team was located on the same floor as the product owner, some meetings were held spontaneously.

7.2. WORKFLOW

Work was mainly done in the office provided by the product owner. This office is on the same floor as the product owner and thus allowed for easier communication. The team would make sure to meet the product owner at least once a week to show their progress. In practice, however, there was much more and often communication between the team and the product owner. This was not an unnecessary luxury, because this project had to be integrated with another project running in the product owner's department as it was being developed. Hence, the direct lines of communication were very beneficial to the project.

Working times were flexible, but approximately from 9:00 until 17:00. At the beginning of the day, the team would have a meeting in which every team member describes what they did the previous day, the team would discuss who would perform which tasks for the coming day and what complications they may stumble upon and how they are tackled.

Also, weekly updates would be sent to the coach via Mattermost to inform him of the progress and to discuss possible problems.

7.3. DEVELOPMENT TOOLS

During the project, GitLab was used as a version control system. Besides version control, it was used for several purposes. One of those purposes was code review. When a team member wants to merge code into another branch, a merge request had to be created. The merge request should shortly describe what the to be merged code is intended to do. Before a merge request could be approved, a team member reviews the code on GitLab. If problems or questions regarding the code arise during the code review, a discussion would be started and a merge request cannot be merged before all discussions are resolved. When all the started discussion would be resolved satisfactory, the reviewer approves. A merge request required at least one approval before merging. This allowed other team members to stay up to date with the code base as well as ensure high code quality. A second purpose is the use of continuous integration (CI). The CI was set up to

ease the development process. It would build the product and perform the tests (see Chapter 5) automatically. If a build contains failing tests the CI will fail and hence prevent these errors from entering the final product. Additionally, GitLab was used as a means of documentation. GitLab's wiki feature was used to document the possible paths of the server and their respective inputs and outputs systematically.

8

DISCUSSION

In this chapter, a reflection on the process will be presented in Section 8.1. Section 8.2 introduces the limitations of the product. Recommendations are given in Section 8.3. Finally, ethical implications will be discussed in Section 8.4.

8.1. REFLECTION

Several issues popped up during the later stage of the project, such as requiring shared memory between processes on the server. These issues could have been solved if it was more clear what the requirements were for the final product. During the research phase, the emphasis was placed on requirements for the initial prototype. This made the initial sprints go smoothly while making the later ones more complicated. During the later sprints, more spontaneous meetings were held with the product owners to discuss the direction of the product. In several meetings, it was noted that the team had a different image of the final product compared to the product owners. This resulted in minor tweaks being required during the sprints, showing the importance of regular meetings and discussions of the current state of the product as well as the future direction of the project.

Some functionalities that were implemented were later to be found redundant. One of such feature is CRC for the connection between the Android device and the Arduino. After the connection was made between these two components, the team was advised to include some error-detection as the communication over the serial port could be flaky. However, after implementing CRC for communication, it was found that the communication is not flaky once established. Only during the start-up, some hiccups would appear.

Adjustments were often required during each sprint, and this is because each time that the scope of the project increased it was found that there were some minor discrepancies between the designed system and the final product. Some of these discrepancies are, for example, how the EPC would be modified. In the initial design, an EPC could be amended directly on the Android device. Later on, when the RFID reader was included in the project, it was known that the communication for these updates would need to go through the RFID reader, instead of directly to the server. Fortunately, these adjustments were usually rather meager because the system is designed to be modular. A misconception in one component would not cause significant issues in the other parts of the system.

It would have been desirable to get access to the RFID reader required for the encoding and decoding of messages earlier. The RFID reader used during the development was an older RFID reader, which did not support the features needed for this. During the last week, the newer RFID reader was made available, and the team got to work with it. Once the encoding and decoding part was tested, the realization was made that the initial design would be too flaky when combined. Thus last-minute changes were required to resolve this issue. The new solution did fix several other limitations that were present in the previous system. These limitations include not being able to handle multiple clients and requests, which was resolved by the use of a unique identifier for each transmission. This way, the server would be able to map each request to a client correctly.

8.2. LIMITATIONS

The current approach is heavily reliant on the accuracy of the encoding and decoding of messages with the RFID tags. This limits the number of concurrent requests that can be resolved. Currently, there is a hardcoded capacity on the server for the number of available unique IDs. This limit is equal to the number of bits of information that will be encoded on the RFID tags, increasing this will result in lower accuracy with the RFID reader. During the development of this product, the numer of bits reserved for data in the encoded messages was 4, this means that the server could only issue up to a limit of 16 unique IDs.

The error detection and error correction method used in this product only allow for correcting a 1-bit error and detecting 2-bit errors. It is possible that the RFID reader would send an arbitrary ID to the server in the case that more bits have been flipped. This could cause issues to the system, as the wrong EPC would be updated with the wrong information. As well as wrong feedback being given to the client. Using the feedback, it can be manually checked that this issue does not occur.

With the current setup when starting the server, the user will need to make an HTTP request to the cache first to start the cache. This is because the server will need to start listening to messages published by Redis when keys expire. Due to Flask not being able to designate one specific worker to take on this task. A workaround was used, where a request would be made to the server to start this service. This request would then be dispatched to a random worker, who will become responsible for dealing with expiring keys in the cache.

During the load testing, it was found that the size of the database could be a possible issue. While it will be unlikely that the size of each request would be nearly 2MB, it is still an aspect that will need to be fixed. This can be resolved using tools that MongoDB provides, for example, sharding would reduce the load. Sharding will make each shard contain a subset of the data, thus distributing the load on each shard.

8.3. RECOMMENDATIONS

The code for the RFID reader interface needs refactoring and testing; the current code for the RFID reader as well as the reader interface require more work to improve the code quality. The code was partially received from the product owner, and not much focus was put into improving the quality of the code afterwards. The connection with the rest of the system got a more leading priority. However, to make sure that the code will be maintainable, it is recommended to test the reader interface and the RFID reader.

The size of the responses from the server could be reduced. Currently, paths often send more information than required. For example, retrieving information of an EPC will return all data of the EPC. It is not possible to only recover parts of the data, which means that the size of these responses will grow over time, as each EPC has several lists to keep track of when the EPC has been read or modified. One way of preventing this is by creating routes that only retrieve either the properties of an EPC or the information about when the EPC was read. This separation would be advisable because the two pieces of information are usually not required together.

The items in the MongoDB could use an index to optimize retrieving the information, and the current setup only uses the id assigned by MongoDB. While this is sufficient for smaller size databases, like the one used during testing, this may result in performance loss when scaling. Adding an index will be beneficial when the database scales to a larger size.

It would be recommended to improve the error detection and error correction even further, one such possibility is by using Reed-Solomon code instead of Hamming code [19]. Reed-Solomon code will allow for the detection and correction of more information compared to Hamming code. However, more research into the application of the Reed-Solomon code in this product will be required.

The reader interface currently sends a request to the server for each tag that has been read in that second. If the number of tags grows, it could exceed the number of requests that the server can handle depending on the configuration for Gunicorn. To prevent it from happening, the requests from the RFID reader could be optimised. One such way is to aggregate multiple tags into one request. This could increase the processing time of that single request, so finding the right balance between the number of requests and the number of

tags per request will be required.

8.4. ETHICS

This project has no ethical implications, and the product does not store or use any sensitive information. Furthermore, the product is intended to be used for research on reducing the costs for RFID systems. This research topic has no ethical implications attached to it either. During the evaluation of the product, the users who have been interviewed are related to the research group, and no confidential information is stored in those sessions.

9

Conclusion

In this project, the architecture for a new RFID system was designed and developed. This RFID system is built to support and demonstrate the research of the Embedded and Networked Systems research group at the TU Delft. The RFID system that was requested would allow information from RFID tags to be stored in a database on the server and requests to modify this information would be done by using an Android application. The communication requires the use of the research performed by the ENS group on encoding information, such that the RFID reader would be able to decode this along with the signals received from the RFID tags.

The final product allows for communicating with the server and storing EPC information on it, as well as modifying this information. The process envisioned by the product owner has been included up to the part of the encoding and decoding of messages. The behavior for encoding and decoding has been mocked using the messages designed during this project. These messages use a predefined syntax that the Android device will use to encode the messages, while the reader interface decodes these messages. Requests can have an expiration time, after which the server will forget this request and send automated feedback to the Android device to inform it of the expiration. Requests that get confirmed by the RFID reader will result in a confirmation message as feedback to the Android device.

Tools such as the dashboard were included to improve the efficiency of performing tests with this system as the dashboard enables the user to quickly add new test data to the database, without requiring knowledge of the server. Furthermore, it also allows users to inspect the data stored and includes an example of possible analytics that can be done on this data.

This product will serve as the foundation of the RFID system envisioned by the ENS group. The designed architecture will allow future scientists to expand the scope of their research to perform measurements on the complete structure.

BIBLIOGRAPHY

[1] "Rfid frequently asked question: How much do rfid readers cost today?" https://www.rfidjournal.com/faq/show?86, accessed: 2019-04-29.

[2] C. C. Aggarwal and J. Han, "A survey of rfid data processing," in *Managing and Mining Sensor Data*. Springer, 2013, pp. 349–382.

[3] S. Binani, A. Gutti, and S. Upadhyay, "Sql vs. nosql vs. newsql-a comparative study," *database*, vol. 6, no. 1, pp. 1–4, 2016.

[4] R. Brown, "Django vs. flask vs. pyramid: Choosing a python web framework," *Recuperado el*, vol. 31, 2015.

[5] X. Cai, H. P. Langtangen, and H. Moe, "On the performance of the python programming language for serial and parallel scientific computations," *Scientific Programming*, vol. 13, no. 1, pp. 31–56, 2005. [Online]. Available: https://doi.org/10.1155/2005/619804

[6] R. Cattell, "Scalable sql and nosql data stores," *Acm Sigmod Record*, vol. 39, no. 4, pp. 12–27, 2011.

[7] D. M. Dobkin and T. Wandinger, "A radio oriented introduction to radio frequency identification," *High Frequency Electronics*, pp. 46–54, 2005.

[8] "Epc™ radio-frequency identity protocols generation-2 uhf rfid," GS1, Brussels, Belgium, Standard, Nov. 2013.

[9] K. Finkenzeller, *RFID handbook: fundamentals and applications in contactless smart cards, radio frequency identification and near-field communication*. John Wiley & Sons, 2010.

[10] C. Győrödi, R. Győrödi, G. Pecherle, and A. Olah, "A comparative study: Mongodb vs. mysql," in *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*. IEEE, 2015, pp. 1–6.

[11] J. Han, H. Gonzalez, X. Li, and D. Klabjan, "Warehousing and mining massive rfid data sets," in *International Conference on Advanced Data Mining and Applications*. Springer, 2006, pp. 1–18.

[12] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1335–1382, 2014.

[13] S. Hatton, "Choosing the right prioritisation method," in *19th Australian Software Engineering Conference (ASWEC 2008), March 25-28, 2008, Perth, Australia*, 2008, pp. 517–526. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ASWEC.2008.22

[14] E. Ilie-Zudor, Z. Kemeny, P. Egri, and L. Monostori, "The rfid technology and its current applications," in *Conference Proceedings of the Modern Information Technology in the Innovation Processes of the Industrial Enterprises (MITIP)*, 2006, pp. 29–36.

[15] M.-G. Jung, S.-A. Youn, J. Bae, and Y.-L. Choi, "A study on data input and output performance comparison of mongodb and postgresql in the big data environment," in *2015 8th International Conference on Database Theory and Application (DTA)*. IEEE, 2015, pp. 14–17.

[16] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empirical Software Engineering*, vol. 10, no. 1, pp. 31–55, 2005.

[17] L. Kumar, S. Rajawat, and K. Joshi, "Comparative analysis of nosql (mongodb) with mysql database," *International Journal of Modern Trends in Engineering and Research*, vol. 2, no. 5, pp. 120–127, 2015.

[18] K. Lei, Y. Ma, and Z. Tan, "Performance comparison and evaluation of web development technologies in php, python, and node. js," in *2014 IEEE 17th international conference on computational science and engineering*. IEEE, 2014, pp. 661–668.

[19] C. Okeke and M. Eng, "A comparative study between hamming code and reed-solomon code in byte error detection and correction," *Int J Res Appl Sci*, vol. 3, pp. 34–39, 2015.

[20] T. E. Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007.

[21] R. K. Panchal and M. A. K. Patel, "A comparative study: Java vs kotlin programming in android," *International Journal of Innovative Trends in Engineering & Research*, vol. 2, no. 9, 2017.

[22] Z. Parker, S. Poe, and S. V. Vrbsky, "Comparing nosql mongodb to an sql db," in *Proceedings of the 51st ACM Southeast Conference*. ACM, 2013, p. 5.

[23] F. Perez, B. E. Granger, and J. D. Hunter, "Python: an ecosystem for scientific computing," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 13–21, 2011.

[24] S. Rautmare and D. Bhalerao, "Mysql and nosql database comparison for iot application," in *2016 IEEE International Conference on Advances in Computer Applications (ICACA)*. IEEE, 2016, pp. 235–238.

[25] L. Rising and N. S. Janoff, "The scrum software development process for small teams," *IEEE Software*, vol. 17, no. 4, pp. 26–32, 2000. [Online]. Available: https://doi.org/10.1109/52.854065

[26] M. F. Sanner *et al.*, "Python: a programming language for software integration and development," *J Mol Graph Model*, vol. 17, no. 1, pp. 57–61, 1999.

[27] K. Schwaber, "Scrum development process," in *Business object design and implementation*. Springer, 1997, pp. 117–134.

[28] T. Shay, "Most popular databases in 2018 according to stackoverflow survey," https://www.eversql.com/most-popular-databases-in-2018-according-to-stackoverflow-survey/, accessed: 2019-04-29.

[29] Q. Z. Sheng, X. Li, and S. Zeadally, "Enabling next-generation RFID applications: Solutions and challenges," *IEEE Computer*, vol. 41, no. 9, pp. 21–28, 2008. [Online]. Available: https://doi.org/10.1109/MC.2008.386

[30] D. Suciu, "Database theory column: Probabilistic databases," *SIGACT News*, vol. 39, no. 2, pp. 111–124, 2008.

[31] R. Want, "An introduction to rfid technology," *IEEE pervasive computing*, no. 1, pp. 25–33, 2006.

A

INFOSHEET

Title of the project: Server Program for retail RFID system
Name of the client organization: Embedded and Networked Systems research group TU Delft
Date of the final presentation: 03-07-2019

Our client is part of the Embedded and Networked Systems group at the Delft University of Technology, for whom an application that is part of their research had to be developed. They are researching new uses of RFID technology and trying to find an alternative solution to the handheld RFID readers as these readers are expensive. Their alternative solution replaces the handheld RFID reader with a smartphone and an additional circuit attached to it.

During the research phase, the best way to put together this system with a fully functioning front- and backend had to be figured out. During the process, some adaptations to the cycle of communications between the different components of our system were made as it changed several times during the weeks, which sometimes caused some of our work to be void. The final product is a working RFID system where interaction with individual RFID tags can be performed without requiring a handheld RFID reader. Some recommendations to improve the scalability and maintainability of the system were made.

MEMBERS

Mike Beijen
Interests: Big Data, Data Processing
Role: integrating RFID reader and microcontroller with project

Kevin Chong
Interests: Algorithmics, Software Engineering
Role: back-end developer, tester

Callum Holland
Interests: Cyber security, Software Engineering
Role: Front and back-end developer

Glenn Keller
Interests: Software Engineering, Human-Computer Interaction
Role: Front-end developer

ADDITIONAL INFORMATION

Client: Przemysław Pawełczak - TU Delft, Embedded and Networked Systems
TU Coach: Mauricio Aniche - TU Delft, Software Engineering Research Group
Contact person: Callum Holland - callum.r.holland@gmail.com
The final report for this project can be found at: http://repository.tudelft.nl

B

Research Report

The goal of this project is to implement a radio frequency identification (RFID) system, that will allow information related to RFID tags to be stored in a database. This data can then be accessed and modified whenever RFID tags are scanned. It must consist of three parts: a server containing information on the available RFID tags with their associated information, an Android application that allows the user to modify the information associated with the tag and a microcontroller (e.g., an Arduino) that serves as an interface to a central reader. As the ENS group would like such a system to built for their use case, the project will start with an initial research phase. The aspects investigated in the initial research phase can be found in this research report, where the problem posed is analyzed, and a possible solution is presented.

B.1. The Problem

In this section, the client will be introduced to understand better what their expectation is, followed by an analysis of the problem posed by the client and finally, an explanation of the relevance of the product.

B.1.1. Client

The client in this project are researchers from the Embedded and Networked Systems group at TU Delft. One of the ongoing research projects related to this one is creating an RFID system that has a lower cost than the current market standards, by removing the need for a standard handheld reader and replacing it with a microcontroller board connected to an Android device.

B.1.2. Problem Definition

The goal of this project is to implement a server program for their research, this program should resemble a retail Radio-frequency identification (RFID) system together with an app that interacts with the server. The use of RFID systems has gained a lot of interest the recently[29]. For this product, the server must be written in Python and accept requests from the app. It must contain a database that stores tag IDs, timestamps, and other information that can be specified by the user. Besides, the server must be scalable in the sense that it must be able to handle multiple connected clients, as well as a large amount of data that needs to be stored. The Android app should be able to retrieve, add, or modify information on the server, as well as allow for communication with an Arduino. It should be easy for the developers to modify and add to the software to allow further extension in the future.

B.1.3. Relevance

Currently, RFID readers are expensive devices with prices ranging from $500 to $2000 [1]. Besides, most RFID retailers sell their products as a complete package, as the software supporting it is dependant on the type of reader (see Section B.5). The ENS group is researching whether the cost of RFID systems could be reduced by using a microcontroller board to interface with the RFID readers, instead of a handheld reader. With such a new approach comes the need for a new software solution; which will be the final product of this project. It will enable further research to be done on their new RFID reader in an environment where all components of an RFID system are connected.

B.2. RFID

In this section, an introduction to RFID will be given. Firstly, the fundamentals of RFID will be presented, followed by an explanation of the Electronic Product Code (EPC).

B.2.1. FUNDAMENTALS OF RFID

RFID is an abbreviation for Radio Frequency Identification and is a technology to transmit data using radio frequencies. An RFID system generally consists of two components: a reader and a tag. The reader is responsible for powering and communicating with a tag [31], while the tag has some information stored on it. Various kinds of tags exist on the market with diverse sizes and formats, they can have different amounts of memory and can be read-only or read-and-write. Once in range, readers and tags can correspond with each other using multiple ways, of which one is backscattering.
As described in [9], using backscattering, power is emitted from the reader into free space. Due to free space attenuation, only a small portion of that power reaches the tag's antenna. That power induces a small current in the tag big enough to power it. A proportion of the incoming power from the reader is reflected by the antenna and returned, again only a fraction of that power reaches the reader's antenna due to free space attenuation. Every antenna has certain reflection characteristics. For transmitting data from the tag to the reader, these reflection characteristics are changed by switching on and off a load resistor connected to the antenna powered by the induced current. By changing the reflection characteristics of the antenna, the amplitude of the reflected power can be modulated, allowing for sending data between the tag and the reader.

In the literature, generally three types of tags are distinguished [7, 9]:

1. *Passive tags* have no energy supply. Instead, they use the radio frequency energy received on its antenna to power the tag. The tag, in turn, can transmit data to the reader, by modulating the RF signal that is received.

2. *Active tags* have a power source, supplying energy to the tag. Having a power supply allows for several advantages such as the possibility of a greater communication range[9] or the possibility of carrying larger memory capacities [7].

3. *Semi-passive or semi-active tags* have a power supply, however, it is not used in transmitting the data from the tag to the reader. The power supply provides power to for example sensors or allows for more memory [14].

B.2.2. EPC

The Electronic Product Code (EPC) uniquely identifies instances of products. EPC is defined in the *EPCglobal Class 1 Generation 2 standard* [8]. An EPC can identify physical objects as well as assets, collections, documents, etc., that all have their own structure. This structure can be distinguished by the different namespaces used. Within computer environments, such an EPC is usually presented in the form of a URI. For example:

```
urn:epc:id:sgtin:0614141.112345.400
```

For transmission, it may be useful to encode an EPC into a binary format. The global standard supports this by providing encoding schemes for different representations of the EPC. The binary encoding of an EPC is stored inside the memory of an RFID tag, which is transmitted when the tag is being activated.

B.3. DESIGN GOALS

In the following section, the design goals of the project will be explained. These goals have been specified at the start of the project and are essential to the success of the product. The important goals are: Maintainability and Scalability, which are further elaborated in this section.

B.3.1. MAINTAINABILITY

Maintainability is an important aspect in this project, as the product will be a component of a larger system. This system does not exist yet and information regarding the whole structure is also limited, thus it should be possible for our product to be easily adapted, to fit the requirements in the future. This also means that the product needs to be maintained by other developers and requires maintainable code, as the final product of this project will exist of several separate components (android app, Arduino, server and database). It will be

important keep these components modular. To achieve maintainable code several sub-goals will be required, namely code quality, testability, and documentation.

CODE QUALITY

Good code quality is important for every software product. During this project, the codebase will have to be uploaded to SIG (Software Improvement Group), the returned feedback on the code will then be used to improve the quality of our code. Furthermore, during the whole process, the project will use peer review to help improve the quality.

TESTABILITY

Testing is important to make sure that the product will show certain behaviour under strict conditions. To make sure that the product will be testable, each component in the system will need to be kept modular. Testing will include unit testing, integration testing, acceptance testing, and performance testing. Unit testing will be required to make sure each individual unit performs as designed. Integration testing will be done between the different components of the system. Acceptance testing will ensure that the product satisfies the requirements of the product owner, this will be important for requirements that are difficult to assess with functional tests. Performance testing will be done to assess the performance of the whole system in different environments.

DOCUMENTATION

For a product to be maintainable, documentation will be necessary. Documentation can ease the process of understanding the code base for new developers, serve as an explanation regarding design choices, and improve the productivity and quality of the developers [16]. Proper documentation will include not only documentation of the code itself, but also documentation of the API. The standard documentation of the code will help with understanding the code. While the documentation of the API will make the development process easier for other developers, which will help with both maintainability as well as extensibility.

B.3.2. SCALABILITY

This product will be mainly used for research purposes, however, to better simulate a practical environment where the product could be used, the product will need to be able to scale properly. The product will be envisioned to be used in a warehouse or a retail environment like a clothing store. Two issues regarding scalability will need to be taken into consideration during development: throughput and storage. Throughput refers to the amount of traffic between the Android devices and the server. Stores or warehouses can have a large number of RFID tags and each interacted with by using the Android devices, while the amount of devices in this system is not limited to one. Storage is another issue as each item will have its own tag which needs to be stored in the database, as well as more information related to this item.

B.4. REQUIREMENTS ANALYSIS

In this section, the requirements will be presented. These requirements were set up by interviewing the product owner and looking at existing products. This is done while keeping the earlier presented design goals (see Section B.3 in mind. The requirements will be presented according to the MoSCoW methodology [13].

B.4.1. MUST HAVES

- The server needs to have a Representational state transfer (REST) API
  - The REST API needs to be able to respond to requests for information related to the EPC of data stored in the database
  - The REST API needs to be able to respond to requests to add new information on tags to the database
  - The REST API needs to be able to respond to requests to update existing information on tags in the database
- The server must have a connection with a database to store information
- The final product must include an Android app

  - The app needs to have a simple user interface
  - The app needs to allow for extension
  - The app needs to be able to send messages to a server
  - The app needs to be able to handle messages received from the Arduino
- The server and the app need to be extensible
- The app needs to be able to communicate with an Arduino
- The server must log all incoming request and outgoing messages

B.4.2. SHOULD HAVES

- The user should be able to explore the state of the database/server through a dashboard
- The user should be able to search within the Android app for specific tags by providing the EPC
- The user should be able to filter the tags on the Android app based on timestamp, EPC, or the fields that the tags have
- The user should be able to obtain more information on the tags such as the type of product, when was it last accessed, price, name or other properties given by the users by tapping on it within the Android app
- The user should be able to define the syntax of the incoming EPC by editing it in-app, without having to change the codebase

B.4.3. COULD HAVES

- The app can save the state of a session, where a session stores all interactions with both the server and the Arduino in the form of a log
- The app can load the state of a previously saved session
- The app can show statistics such as a difference in quantity of items or the numerical proportions of each category of items obtained by comparing different sessions

B.4.4. WON'T HAVES

- The messages will be encrypted during communication

B.5. EXISTING PRODUCTS

To get a better grasp of what the use of the final product would be, research into existing products was done. Some of the existing products found during this research that will be highlighted below are: *AVEA* [1], *U Grog It* [2], *Datex* [3], and *rfidSystem.pub* [4].

AVEA

AVEA is a company that create and manufacture RFID products, one of the products they created is an "IoT RFID Reader". The readers support access to the information related to an RFID tag by using HTTP requests.

U GROK IT

U Grok It is an RFID platform that provides RFID systems to organizations of all sizes. Its reader is a handheld reader called *grokker* that attaches to smartphones. The *U Grok It* app can be used to identify and track items as with other supply chain solutions.

[1] https://avea.cc/
[2] https://www.ugrokit.com/tech.html
[3] https://www.datexcorp.com/hardware/mobile-computing/
[4] https://github.com/gmalsack/rfidSystem.pub

DATEX

Datex is a company that offers full-service solutions to provide supply chain software and mobile computing solutions. Using its mobile computers, of which some also allow reading RFID tags, the company caters complete supply chain solutions. Datex provides complete RFID solutions for warehouse management, compliant with their mobile computers.

RFIDSYSTEM.PUB

rfidSystem.pub is an open-source RFID system designed to be used with the readers from *AVEA*. *rfidSystem.pub* uses the readers to get information on employees and their access to rooms. The access to the rooms is controlled by sending information of the tags to a server.

However, these existing products do not fit the requirements set for two main reasons: accessibility and flexibility.

ACCESSIBILITY TO THE CODEBASE

rfidSystem.pub uses the GPL-3.0 [5] license, which means that the code can be used, modified and distributed. However the same does not hold for *U grok it* and *Datex*, these two products are sold on the market and the public does not have access to the code of the software used. This means that the user would not be able to inspect the code and configure it as they wish.

FLEXIBILITY OF THE RFID READER

As mentioned in Section B.1, a product that will be able to use an Arduino as a microcontroller is required. However, all of the products mentioned above require the use of specific readers and would thus not work with the envisioned setup. While in the case of *rfidSystem.pub*, the implementation could be adapted to allow for this connection as it is open-source, the product has its disadvantages. Even though *rfidSystem.pub* is a small project that could suit the requirements, documentation of this project is lacking. It would be more desirable to create our own product with proper documentation.

B.6. DESIGN CHOICES

In this section, the motivations behind several design choices made during the project are given. Including the design choices for the server backend, database and android app language.

B.6.1. SERVER BACKEND

As for the server, it was requested to use Python. An important choice for the backend was made regarding the web framework. Several frameworks were considered, with Flask [6] being chosen in the end.

LANGUAGE

The server in the backend runs on Python, as this was requested due to the readability of Python. As Python has a simple and clean syntax [20, 26]. On a more technical note, Python suits this project for several reasons:

- Python comes with a lot of useful community libraries [20]
- Python is efficient for scientific programming and high-level scientific software development [5, 23], due to the libraries created by the community

WEB FRAMEWORK

Flask is used for the web framework, as it is more suited for small projects compared to other frameworks like Django [7] and Pyramid [8] [4]. Because, Flask is a micro framework, while both Django and Pyramid are full-stack frameworks. This means that Flask only includes a minimal amount of modules, including modules that are required in this project, such as request/response handlers. While full-stack frameworks provide more modules, this can hinder the development process when these built-in modules have to be avoided.

[5] https://www.gnu.org/licenses/gpl-3.0.en.html
[6] http://flask.pocoo.org/
[7] https://www.djangoproject.com/
[8] https://trypyramid.com/

However, Flask by itself is not scalable and not intended for production, as it can only handle one request at a time. For this reason, another framework, Tornado, was considered, however, Tornado is less well known and thus has less information that can be found, this will be undesirable for developing and maintaining the product. In the end, the decision was made to use Flask with Gunicorn [9] for deployment, this will solve the scalability issue of Flask, while keeping the benefits of Flask.

Node.js [10] is another language that was taken into consideration. As Python does not support multithreading it can cause scalability issues, for this reason, Node.js would be a better solution in high concurrency situations [18]. However, this issue is resolved by the use of Gunicorn, this will allow for using multiprocessing to handle the requests to the server.

B.6.2. DATABASE

Three types of databases were considered for this project: SQL, NoSQL, and probabilistic databases. The decision was made to use a NoSQL database, MongoDB, for the following reasons:

NOSQL VS SQL

The decision to use a NoSQL database over an SQL database was made because of several reasons, namely dynamic schema, scalability and speed.

Dynamic Schema

In relational databases, the schema will need to be defined beforehand, while in NoSQL databases the schema can be dynamic. This means that fields can be added to the database later on with little to no effort. As mentioned in Section B.3.1, this product will be part of a larger system, and thus it is not yet clear what the schema will be and should be kept as extensible as possible. In this case, having a dynamic schema means that the database will be able to satisfy this requirement, compared to using a relational database, which requires the schema to be defined beforehand and does not allow for efficiently changing the schema [22].

Scalability

The amount of data stored in a database dealing with RFID can be enormous [11], thus scalability of the database is an important aspect to take into consideration. Traditional SQL databases like MySQL provide vertical scalability (with limited horizontal scalability with master-slave replication). Vertical scalability means that the performance can be improved by adding resources like memory and processors. On the other hand NoSQL databases like MongoDB provide horizontal scalability, which means that the performance can be improved by increasing the number of machines to share the load [3, 24]. While scalable RDBMS do exist (e.g. MySQL Cluster), all operations and transactions require to be performed on a small scale [6].

Speed

In [10] it is shown that MongoDB has a lower querying time up to two orders of magnitude compared to MySQL for all CRUD (Create, Read, Update, Delete) operations when working with a database with a size of 10000. The same was observed in [17], where a performance test was made with 100 to 50000 entries. Another comparison was made between MongoDB and PostgreSQL in [15], where the execution times of INSERT, UPDATE and DELETE operations of MongoDB were faster than that of PostgreSQL. While MySQL and PostgreSQL are not the only possible representatives of SQL databases, they are two of the most popular databases [28].

PROBABILISTIC DATABASE

Another type of database that was considered was probabilistic databases, a generalization of relational databases that includes a notion of uncertainty for the data it stores. This means that the database can deal with imprecise and uncertain data [30], which is ideal for RFID systems as RFID data is prone to errors [2]. However, the use case for probabilistic databases as mentioned in [2] is mainly for high-level event extraction, where RFID data is used to identify events that are a sequence of specific readings. In those cases, a notion of uncertainty regarding the occurrence of an event is important. In this project, this is not as relevant, as the product will currently be focusing on dealing with independent readings.

[9] https://gunicorn.org/
[10] https://nodejs.org/en/

B.6.3. APP LANGUAGE

To design the Android app the decision was made to use Java, rather than Kotlin. The choice was made due to the larger existing support for building Android apps in Java compared to Kotlin. While researching both Java and Kotlin, several advantages for Kotlin were mentioned in [21], however, these advantages do not outweigh the benefits of Java's greater support community and the team's familiarity with Java.

EASY TO LEARN AND WRITE LESS CODE

One of the advantages mentioned in [21] was that Kotlin is easier to learn and you have to write less code, thus making it more readable. However no experiments regarding readability or ease of use were performed, thus these claims are not backed up by any data and no conclusion can be made regarding the difference in readability and ease of use between Kotlin and Java. However, as per the requirements, the code had to be easy to read and Pythonic features have been given as an example (e.g. nested list comprehension), usage of this should be limited as it makes the code more difficult to comprehend for less experienced developers. Kotlin in the same sense reduces the amount of code that needs to be written by removing the need for boilerplate code.

NO **NULL** POINTER EXCEPTIONS

The way Kotlin provides Null safety is by not compiling code that can potentially cause NullPointerException (NPE). This is indeed an advantage over Java as it provides the guarantee that the system will never crash due to NPEs. Even though this guarantee cannot be obtained in Java, by thoroughly testing the product, the chance of NPEs appearing should be mitigated.

INTERCHANGEABILITY WITH JAVA

This is not an advantage of Kotlin alone, as the option exists to convert Java to Kotlin, but also Kotlin to Java.

B.7. ARCHITECTURAL VIEWS

In this section, the envisioned software architecture of the product will be described. This will be done by presenting a subsystem decomposition as well as a sequence diagram.

B.7.1. SUBSYSTEM DECOMPOSITION

Figure B.1 shows the envisioned subsystem decomposition. A short elaboration on its elements will follow in this section.

The *Arduino* is the microcontroller in the system that serves as an interface with the central reader, in this product its purpose is to provide the Android device with the EPC tag. It communicates the tag data to the Android app using the UsbSerial library [11]. For the scope of our project, this can be fictitious data (see B.1). The Arduino also needs to be able to receive messages for establishing connections and receiving commands from the Android device.

The *Android app* sits between the server and the Arduino. It takes care of the messages received from the Arduino and creates HTTP requests that the server can handle. It can show information about the tags that have been read such as earlier appearances, locations, etc. Besides, the app may also be used for updating meta-information about tags or to perform searching or filtering on the database and displaying these results in an understandably and intuitively way.

The *Server* is the crux of the project. It receives HTTP requests from the Android app and turns them into queries for the database. Depending on the request, the server may answer with either an acknowledgement or the result of the query.

The *Database* is a MongoDB database, its function is to store all the data related to the RFID tags, such as the EPC and the properties of the item that the tag is attached to. The database only communicates with the server and it will execute the queries received from the server to add, retrieve, update or delete entries. Depending on the result of the query it will respond to the server with the result or some acknowledgement.

[11] https://github.com/felHR85/UsbSerial/

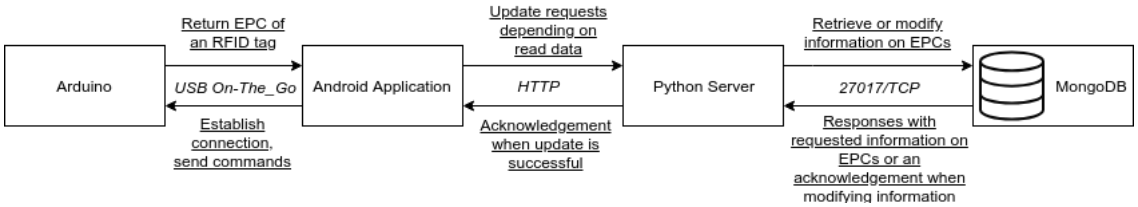


Figure B.1: Envisioned subsystem decomposition of the product

B.7.2. SEQUENCE DIAGRAM

Figure B.2 depicts a possible interaction with the final product. A user presents an RFID tag to Arduino. The read EPC is sent to the app, which then sends it via an HTTP request to the server to update the EPC's presence. The server updates the database. Then, the query is successful and it is acknowledged to the user.

Having added an appearance to the database, a user may want to receive further information about the tag such as earlier appearances. Upon request, the app will then send the EPC to the server with a request for the information on this tag. The server will query the database for all the information about the tag, and the database will respond with that information to which it may do some post-processing. Finally, the information is shown to the user.

As can be seen, the function of the app is twofold. On the one hand, the app interacts with the Arduino to retrieve information about the read tags. On the other hand, the app can be used to query the server concerning more information about EPCs.

B.8. DEVELOPMENT METHODOLOGY

In this section, an introduction to the used development method will be presented.

B.8.1. METHOD

During this software project, Scrum [27] will be used as the main method for development. Sprints were chosen to be one week, where each sprint ends with a milestone that has a piece of working software. Sprints will start with a sprint planning, where the features for this sprint will be determined. Daily stand-ups will be held to discuss what everybody has done, what everybody will do and if any issues need to be resolved. At the end of each sprint, there will be a sprint review and evaluation. One of the reasons for choosing scrum are the sprints that will improve the process of developing a product that will satisfy the product owner, as well as allow for the possibility to adapt to any changes in the requirements for the product, while still enjoying a very clear and transparent systematic approach.

B.8.2. COMMUNICATION

As the team has access to an office that they work in from 9:00 until 17:00, most communication will take place face-to-face. In case any team member is absent or somehow unavailable, communication can take place via Mattermost. Communication with the coach and product owner will also take place via Mattermost. The team has a weekly face-to-face meeting with the product owner to inform him of the progress made by the team.

B.9. CONCLUSION

To conclude the research phase, the goal of this project is to build an RFID management system that can handle communications between an Arduino and an Android device, and between an Android device and a server to store or modify some information in a database. The proposed solution is to use Python with Flask for the server, Java for the android app and MongoDB for the database.

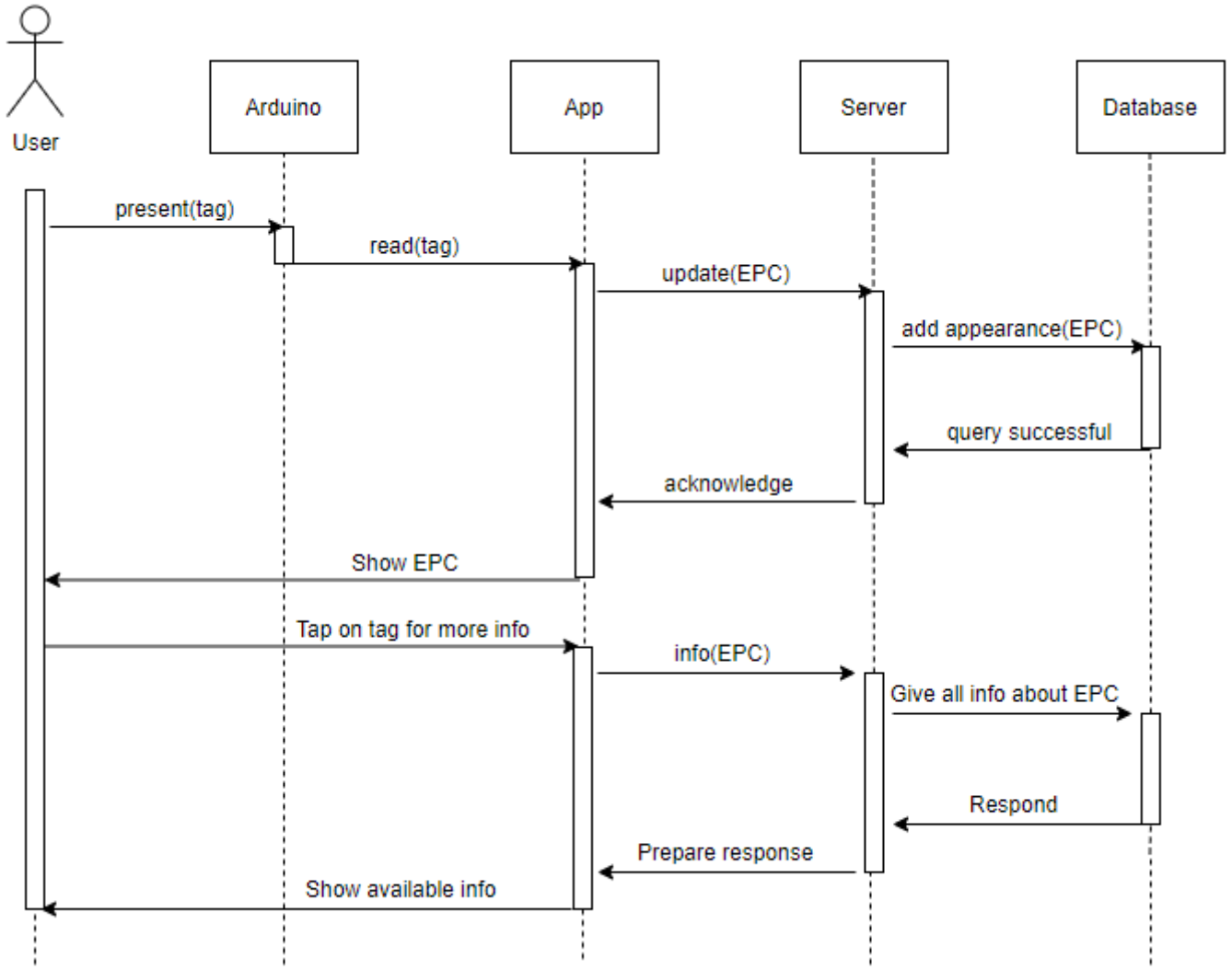


Figure B.2: Sequence diagram of a possible interaction with the final product

C

Original Project Description

The adoption of RFID systems in modern retail environments can increase productivity and inventory management. However, each implementation of an RFID system requires a customised software solution dependent on the requirements of the client. A backend server program with database, message handling and easy access is required for seamless integration.

The Embedded and Networked Systems group (www.ens.ewi.tudelft.nl) at TU Delft requires such a program for continuing research in retail RFID. This means that the program needs to be written in an easy readable language (Python preferable) and in a way that allows for extension in the future.

Furthermore, an Android app has to be written that can communicate with the server. As with the server program this app has to be written with extension possibilities in mind.

To summarise, the goal of the assignment is to write the following two programs:

- *Server program* (preferably Python [version 3.6 or higher]):
  - Needs to contain a database that contain tag ID numbers and additional data linked to s given tag;
  - Requires message handing capabilities, i.e. messages for the user or status/data for system;
  - Needs to handle multiple connections, i.e. multiple client phones and multiple RFID readers (reader programs are out of the scope of this assignment, however the server needs to handle simple messages containing tag information, i.e. [tag id: some data])
- *Android app*:
  - Requires simple GUI that is easy to add to and extend. Needs to at least contain basic “send message to server (with receive confirm)” button;
  - Requires connection which allows for message (data/status) sending and receiving from and to the server

D

Requirements

D.1. Must Haves

- The server is able to store information related to existing EPC in its database, this information includes the EPC itself, timestamps to show when this EPC was accessed, and other information that can be specified by the users.
- The server is able to update information on existing EPC in the database.
- The server is able to check whether an EPC is contained in the database.
- The server is able to store a request from another machine, this request will be used to indicate that the machine has requested an EPC.
- The server is able to respond to the request mentioned above depending on the circumstances within a certain duration after the request has been made. This duration serves as an expiration time for the request.
  - If another machine confirms that an EPC has been read within this duration, the server will respond with the EPC that has been read.
  - If no confirmation is made during this duration, the server should respond to the requester that their request has expired.
- The Android application is able to send a command to the microcontroller, this command is used to indicate that the microcontroller should power the magnetic coil attached to the microcontroller.
- The Android application is able to send a message to the microcontroller in the form of a string.
- The Android application is able to notify the server that it would like to make a request.
- The Android application is able to receive the response from the server, to get informed of the result of the request.
- The Android application is able to show the user the information on the EPC that the server has returned, if the request was processed successfully.
- The Android application is able to restart the requesting process, if the previous request has failed.
- The microcontroller is able to receive a signal from the Android device to start powering the magnetic coil attached to it.
- The microcontroller is able to receive a message in the form of a string from the Android device.
- The interface for the RFID reader is able to get all read EPC tags and their related information from the RFID reader.

- The interface for the RFID reader is able to send the information on the tags that it has read to the server.
- The interface for the RFID reader is able to send a confirmation to the server that a tag has been read recently.

D.2. SHOULD HAVES

- The server is able to handle at least 350 requests per second, as this is the amount of tags that are read per second by the RFID reader when using 5 tags.
- The server should include a web page that serves as a dashboard for the server. Where the state of the server can be inspected.
- The server is able to process requests for transmission from multiple clients.
- The user is able to configure the settings of the Android application to connect to the server and the microcontroller without requiring changes to the code.
- The microcontroller is able to detect errors in the messages received from the Android device.
- The microcontroller is able to encode a message in the data transmitted from the RFID tag.
- The interface for the RFID reader is able to decode a message received in the data transmitted from the RFID tag.
- The interface for the RFID reader is able to

D.3. COULD HAVES

- The server could include options to import the information on EPCs stored in the database.
- The server could include options to export the information on EPCs stored in the database.
- The server could include visualization of the information stored in the database, such as how often an EPC was found over time.
- The Android application could automatically restart a transmission if the previous one failed, this can be done a few times until the transmission succeeds.
- The RFID reader interface could notify the server when the received message had an error.
- The microcontroller could repeat the transmission process several times to increase the chance of succeeding.

D.4. WON'T HAVES

- The RFID reader interface will not store any information on the tags that it has read previously.
- The server will not include measures to improve the security of the system, because this product will be used internally for research. Input validation will only be used to check for correct parameters.
- The system will not be tested with other microcontrollers than Arduino Nano.

E

SOFTWARE IMPROVEMENT GROUP FEEDBACK

E.1. FIRST FEEDBACK

De code van het systeem scoort 4.6 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere deelscore voor Unit Size.

Bij Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Dit kan verschillende redenen hebben, maar de meest voorkomende is dat een methode te veel functionaliteit bevat. Vaak was de methode oorspronkelijk kleiner, maar is deze in de loop van tijd steeds verder uitgebreid. De aanwezigheid van commentaar die stukken code van elkaar scheiden is meestal een indicator dat de methode meerdere verantwoordelijkheden bevat. Het opsplitsen van dit soort methodes zorgt er voor dat elke methode een duidelijke en specifieke functionele scope heeft. Daarnaast wordt de functionaliteit op deze manier vanzelf gedocumenteerd via methodenamen.

Voorbeelden in jullie project:

- $cache\_api.py : add\_to\_cache()$
- $main.py : setup\_logging()$

De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid testcode ziet er ook goed uit, hopelijk lukt het om naast toevoegen van nieuwe productiecode ook nieuwe tests te blijven schrijven.

Over het algemeen scoort de code dus bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.

F

RESULTS OF LOAD TESTING

# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
435	53	71000	75505	1164.853811264038	169200.0229358673	1598222	2

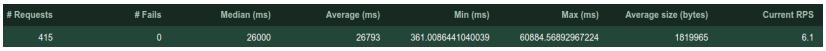
Figure F.1: Flask with 10000 items in the database

# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
415	0	90000	91203	1499.992847442627	191885.46085357666	1819965	1.9

Figure F.2: Flask with 10000 items in the database

# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
433	0	25000	27156	358.7172031402588	64945.46437263489	1819965	6.3

Figure F.3: Gunicorn with 10000 items in the database



# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
415	0	26000	26793	361.0086441040039	60884.56892967224	1819965	6.1

Figure F.4: Gunicorn with 10000 items in the database