University College of North Denmark

Bachelor - Computer Science

Project Report

Draganoiu George-Alexandru

Supervisor: Nadeem Iftikhar

Submission date: 24/10/2018

Contents

Problem statement

Obi Plus is continuously expanding. Always trying to integrate more services and solutions in their way of becoming one of the leaders in internet connected vehicles. One of their visions is the ability to pay for parking via mobile phones.

To achieve this vision multiple different platforms and devices must come together. For this to work the user has to have a OBD device attached to his car. The system could detect the user in a parking lot, by overlapping the device’s location with the location of parking lots in the system. Once confirmed, the user could then pay for parking before leaving the car just by using the phone

By integrating the parking payment, the client can have a more seamless user experience every time he pays for parking by eliminating the steps of manually paying for parking.

For this project to become a reality the companies and/or the municipality must accept it as a valid way of payment and the parking lots themselves have to have their coordinates measured and stored in a database. The purpose of this project is to show that this system can be implemented but at the same time leave room for further features and improvements.

Methodology

Summary

A methodology is a collection of activities that dictate the production process that leads to the development of a software system. There are multiple types of software systems and there is no universal methodology that can be applied to all. The methodology practices depend on the category of the software being developed, the requirements both of the project and of the customer, the team inside the company and the structure of the company.

There are different types of methodologies but they all have the same 4 main components that are at the center of any software engineering development:

1. *Software specification.* This step focuses on gathering all the requirements and the measurements that are necessary for the development of the product. This also becomes the agreement between the customer and the development team of how the final product should function.
2. *Software development.* This step uses the aforementioned requirements do develop the product.
3. *Software validation.* At this step, the software product must be measured against the customer's needs.
4. *Software evolution.* This step is ongoing and it focuses on meeting future requirements and maintenance of the current developed software.[1]

These 4 fundamentals of every methodology can, in turn, be divided into further subdivisions. Some examples include functional requirements, domain models and unit testing. Methodologies also include other activities that are directly related to managing and implementing the specific methodology. The way a methodology is described focuses on the activities involved in the process, their order and the result produced out of each step.

Methodologies can be characterized by their 3 main components. The conditions, the roles and the products. Every methodology has a certain sequence of actions. For each action to take place there must be certain pre and post conditions that let the engineers know that the action should start or end. For example a testing activity must have as a precondition the requirements of the software and one of the post conditions can be a certain code coverage. Each methodology has its roles for the people involved with the product development. Each role has its own tasks that fulfill a specific need. Some examples are managers, programmers and testers. Lastly, each activity should have a distinct purpose. Therefore, after each activity, a product or deliverable should be expected. These outcomes can be diagrams, requirements, etc. [1]

Different companies use different methodologies based on their needs and capabilities. Safety-critical systems require a step-by-step and rigid development process while smaller companies that churn different software solutions require a more flexible, agile methodology.

To choose the right methodology, multiple models must be considered and compared. These are high-level abstractions that can be later adapted to create a more specific engineering process. The first model to talk about is the waterfall model.

Models

The waterfall model

The waterfall model follows closely process models used by military systems engineering from the 70s. It is one of the first ever models introduced to software engineering. As seen in the figure above the waterfall model phases cascade from one into another. Because of the model is designed, each step happens only once. In principle, each step is planned before the start of the project and carried out. Because planning is so important, that makes this model a plan-driven model.

The waterfall model has 5 steps. Each step can be traced back to the 4 basics of software development. The only difference is that the software specification is divided into the first 2 steps of the waterfall model. The steps are *Requirements, Design*, *Implementation, Verification* and *Maintenance*. [1]

The waterfall model is used in projects that have clear specifications or that contain critical systems. It has clear boundaries for each stage. At the end of each stage, one or more documents need to be approved. Some systems benefit from this approach:

1. Embedded systems where the software interacts directly with the hardware. The rigid structure of the hardware system makes changes in the requirements improbable, thus delays are minimized.
2. Critical systems that are required to abide certain levels of safety and security benefit from an extensive documentation. For the system to work the requirements and analyses have to be done in detail.
3. Large systems that are built in tandem over different departments or companies. This requires extensive documentation to keep everyone up to date and on the same page. [1]

The waterfall model has however some problems. Because the stages don’t overlap, it makes the model very rigid and difficult to adapt to changes. In software development it is normal to have stages communicate between them. This solves problems for earlier stages that are discovered in later stages. If the requirements aren’t understood or they prove to be too expensive, that may lead to a lock on the project or even a shutdown. On the other hand, if the system is deemed too easy to add new features will mean to restart the process and that will drive the costs up.

Another problem this model poses is that the product is delivered in its entirety at the end. If there were misunderstandings in the development process, it may be too late or too expensive to change anything. Lastly, testing happens late in the process, after the implementation stage has ended. If any serious issues are discovered, there will be expensive to fix.

Incremental development

Incremental development has the opposite approach. It focuses on intertwining the stages of development and on feedback across those stages. As the system is developed incrementally, the feedback can come from both the client and the users. Incremental development is the standard for most systems that are developed today. It can be both plan-driven and agile, or just a mix of both.[1]
Agile increments contain the functionality that is highest on the customer's list at that time. This helps make sure the customer gets what he needs while adjusting for new functionality in future increments.
The incremental development model has some advantages over the waterfall model:

1. By accepting change and reducing the amount of work that has to be redone, the cost of changes is reduced.
2. By delivering the software in increments, it is easier to get feedback from the customers or users. This ensures that the customers get the program that they need.
3. By delivering the most important functionality first, the team can deploy the system early even if it is missing some of the functionality. That way the customers can get value sooner.[1]

There are also downsides to the incremental development process:

1. In large companies it can come in conflict with the overall structure of the company. Some companies require documentation as primary form of communication between teams.
2. The development process is more loose and harder to measure and quantify. That will make a manager's job harder and development issues that are not caught due to lack of documentation can reoccur.
3. Certain systems can not be easily divided and require multiple increments before any part of the system is tested by the users. The lack of feedback will reduce the value of the increments.
4. By adding many changes and features to the original requirements, the system can become cumbersome. Because of this, frequent refactoring becomes crucial.[1]

Model selection

Looking at the project at hand, there seem to be multiple platforms that need to be integrated. Data that is sent from the OBD (On-Board Device) needs to be captured and converted. That information then needs to be siphoned and sent to the appropriate part of the system. From there the necessary checks should result in a message being sent to the owner of the device. By taking into consideration all the different platforms the system should integrate, the best approach would be an incremental model.

The integration between these platforms can change over time and some platforms can be changed for other based on the needs of the customers or incompatibilities that are not yet discovered. Having the ability to build the system in increments without having to go back on every change is beneficial. By designing while building, problems can be dealt with in real time and minimum cost. Intertwining the development stages will fix unforeseen issues caused by integrations. At the same time, having a tool such as refactoring will clear code as new code is added.

That comes with some drawbacks. If the development process does not provide enough structure and documentation, it can degrade and lead to resources poorly allocated. The system only has one functionality, that is the product of multiple integrations. This makes it difficult to have an early delivery of a functional subset of the system. This will undermine some of the benefits of incremental delivery such as the value of having feedback and leave possible errors for late discovery.

Methodologies

Agile vs. plan driven

Incremental development can be done by using a methodology that is either plan-driven, agile or a combination of both. Unfortunately, many of the software systems developed today do so in a very dynamic environment. Requirements can change at any step in the development process and companies need a way to rapidly adapt. A plan driven development will have the requirements set in from the start. That takes away from the adaptability power of the incremental model. For that, agile principles are used.[1]

As mentioned before, agile emphasizes face-to-face, direct communication with the customers, product managers and developers amongst others, over detailed written documentation. Thus allowing the product to be able to morph based on changing needs. It also focuses on rapid delivery of software.

Most agile methods structure themselves, in a way that they have small time frame, called iteration, which is typically anywhere from 1-4 weeks long. At the end of each of them, it is up to team to evaluate the situation and prioritize what to do next. For this, the team employs a set of tools that come in support of the development process. Tools such as, but not limited to, automated testing, progress tracking and management support. [1]

Unlike plan driven, agile measures progress based on working implementations and/or fixes of the system rather than following a strictly structured plan. Mixed with its iterative nature can make it quite inefficient in a large company, where most employees would be accustomed to the former. Another drawback is the lack of proper documentation, which makes it extremely hard for latecomers to get in touch with the whole process. This can also lead to derailment of the original requirements, which in of itself, runs the risk of increasing the cost.

This, in practice, is solved by adding some of the processes found in plan-driven development. Agile increments can also be used for plan driven activities, such as requirements and design, to produce the necessary documentation. This documentation can be used then by management to have a better overview of the progress.

There are multiple agile methodologies that have developed over the years to serve the different needs of software development. Some of the most used ones today are Scrum, Kanban and Extreme Programing.

One of the first agile methodologies developed was extreme programming. It creates scenarios about the system that needs to be developed. These scenarios can then be easily understood by non developers and give the developers a clear idea of what the requirements are. These scenarios, also called user stories, are then prioritized and added to a backlog from where the developers can select the most crucial ones.[1]

It follows the agile principle of developing functional code. This principle is defined by some of the XP principles; for example simplicity with small changes. These small changes keep the code modification to a minimum. To help keep errors to a minimum, pair programming and collective ownership of the code are used. One of the more unique principles that it follows is that of test driven development.

One of the most used agile methodology today is Scrum. It has roles, ceremonies and artifacts that allow the development of software in increments called sprints. The unique aspect of Scrum is that it doesn't define how the software should be developed, allowing the use of other development patterns. For each sprint, tasks are taken from the project backlog, discussed and allocated. During the sprint, daily meetings are held to discuss progress. At the end of the sprint, another meeting is held to measure progress and review performance. This data is then used to calibrate the next sprint.[1]

Methodology selection

Choosing scrum for this project allows for an agile framework to be combined with some UP practices. For a project of this size agile can provide a rapid development while UP can add structure. Adopting Scrum as a general process for developing the system. Implementing a backlog. The backlog contains the list of all the functionality desired, in the form of user stories. These user stories describe every step of the interaction between the user and the system. The stories are then developed based on their priority.

Diving the remaining 6 weeks into 3 2-week sprints will allow sufficient time for sprint planning, sprint retrospective and provide enough time for code to be worked on uninterrupted. A work in progress limit will be imposed on the number of tasks the developer can work on at a given moment. The development will be done in small iterations that will be continuously integrated into the main project. These iterations need to be tested before integrating and refactoring needs to be done after.

To add structure to the project, diagrams were imported from UP. These will help edify the development process and keep the developer on a clear path. The system does not have a lot of interaction with the user but behind the scenes there are many interacting components. Because of this, an interaction diagram will provide a clear documentation of the desired sequence of interactions from the device to the user.
The domain model can render the conceptual entities and the relationships between them. This will help the developer have a clear mental map of the system and can weed out potential relational problems. The design diagram is the last diagram that provides a detailed representation of the functions of each class.

Requirements

Functional

Based on the problem statement and the developed use case, we identified several functionalities that the system should have.

1. The ability to park a car in a designated parking area.
2. The ability to accept or deny the notification.
3. If accepted, the app to go to the mobile pay.
4. The ability to pay for parking time.
5. The ability for the user to register and log in the system.
6. The ability to automate payments .

Non-Functional

Usability

The system should be designed in such a way that most of the work should be handled in the background and the user should have minimum interaction.

1. The user should get a notification when stopping the car in a designated parking area.
2. The user should be able to proceed to payment in as few clicks as possible.

Reliability

The system should always be available and be able to handle all interactions with a high accuracy.

1. The system should reliably determine if the user is in a parking area.
2. The system should still function even if third party software is not responsive.
3. The system should have high uptime.

Performance

The system is expected to have a high number of users, however due to the nature of the system, a fast response time is not expected.

1. The system should be able to handle fluctuations in the number of requests based on the time of day.

Supportability

From this point of view, the system should be testable and easy to maintain.

1. The system should log important messages.
2. The system should be testable.
3. The system should be modular.

Design

Sprint 0

In order to initiate the project some planning is required. This is usually done before the start of the development and it is called sprint 0. During this phase the architecture is structured and the work is divided into iterations. At this point the working environment is set up based on the system requirements and team specializations. This includes frameworks, development operations and the skeleton of the system.

Looking at the problem statement, there is a device that connects to the car and transmits data about location. That data then needs to be analyzed, processed and interpreted. At the same time, the device sends many different signals that will also need to be analyzed and processed. The system should also be able to process messages from multiple devices with ease. Because of this, the system should be built in a modular way that would help scalability. To achieve modularity, the main tasks of the system should be separated.

Use case	
Pre-conditions: The user had the engine on.	Post-condition: The user receives a message of payment.
Typical course of events	
Actor	System
1. The user arrives at a parking lot that has the mobile payment feature 2. The user turns off the engine	3. The device is triggered by the engine turning off 4. The device sends the event corresponding to the engine being turned off 5. The system receives the message 6. The system identifies the message and decodes the message 7. The system acknowledges the message 8. The system identifies the user and the location 9. The system sends a message to the user

Table 1 - System use case

Scenarios that result in the system failing:
1.a If the user arrives at an unmarked parking lot there will be no interaction.
2.a If the user does not turn off the engine the device does not trigger.
3.a/4.a The device malfunctions
5.a The system is offline

5.b The system does not receive the message
6.a The message is corrupted
7.a The system fails to respond.
8.a The system fails to identify the user
8.b The system fails to identify the location
9.a The system does not send the message to the user

To capture that data and turn it into useful information, the python language was selected. It is open source and it provides a low level networking interface module that the device can connect to. Because the job of the python code is to just convert the data of the device, it is virtually stateless. That means that, based on the traffic and number of devices, multiple python modules can be run to cover the workload. This provides an opportunity for scalability.

To capture the information transmitted by the python module, a service bus will be used. The service bus will ensure that no matter the workload, all messages will be handled by the appropriate service. This ensures that the system is scalable and can handle growth.

The last part of the system will take the information related to the parked car and query a location database. Based on the result, it should either ignore it or inform the users that they are parked in a parking zone. By implementing this as a separate service and decoupling the system parts, it allows for further growth of the system.

A NoSQL database was chosen for storing and querying locations. This comes with some benefits. NoSQL is typically faster and it allows clustering on the location data itself. It also allows for easy alterations and since the data is static, it also allows for easy scalability. A second relational database is recommended for identifying the user via the device id. This identification can serve for many other areas of the system that are out of the scope of this project. A general architecture of the desired system can be seen in Fig. 1 below.

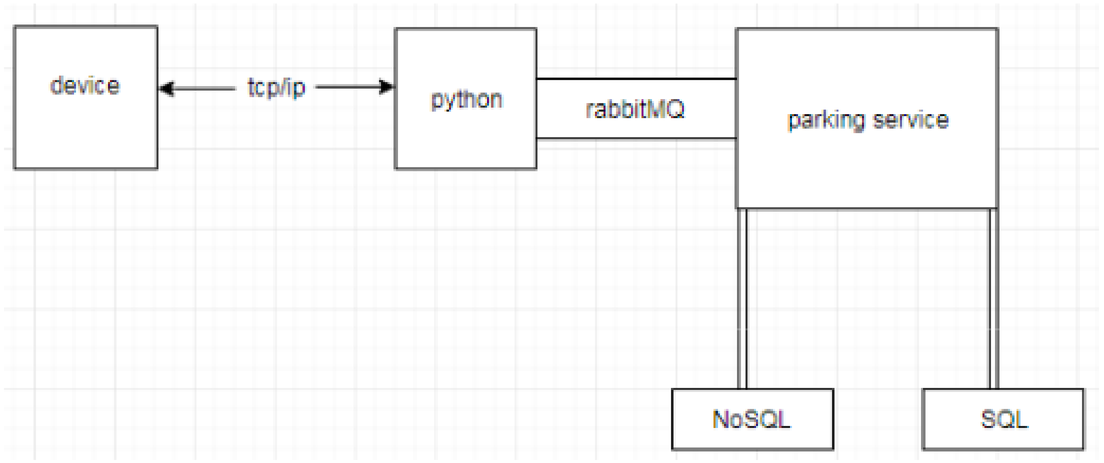


Fig. 1 - System architecture

In order to validate the work done during a sprint, testing is used. To pass the sprint life cycle, unit testing will be used as a way to ensure proper functionality of the tasks developed. To help with future development and maintenance, logging is used to report any malfunctions. To implement incremental development, continuous integration was chosen to keep track of development and possible building problems. Lastly, to create a backlog for the software development sprints, trello was chosen. It is free and provides high customizability. All the tasks go in the backlog list, and for each sprint, a number of tasks are chosen to go in the tasks list. These tasks then have to be developed and tested before being considered done. If the time allocation does not match and tasks are left, at the sprint review, alterations are made and the tasks are re-added to the backlog. Alternatively, if tasks are completed before schedule, then the next most important tasks are added from the backlog.

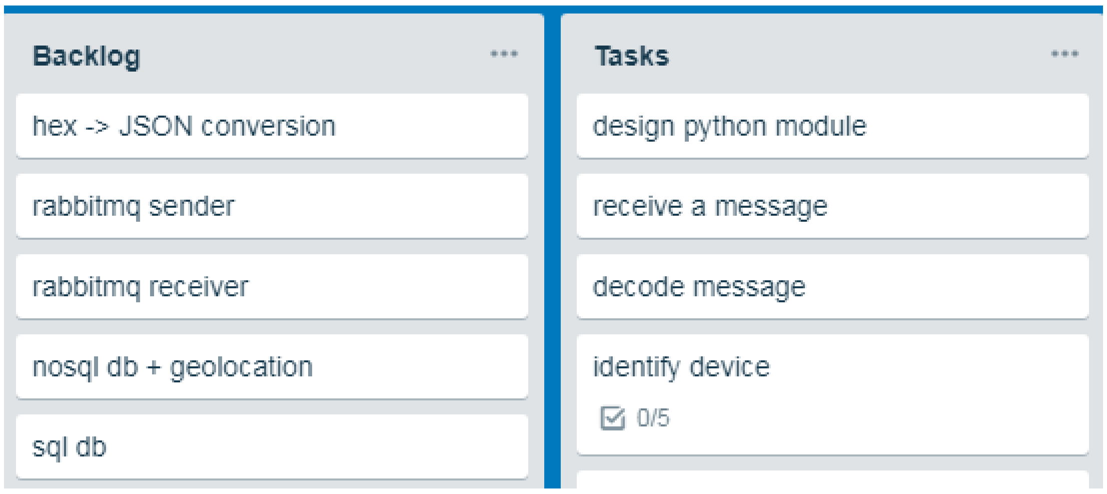


Fig. 2 - Backlog 1

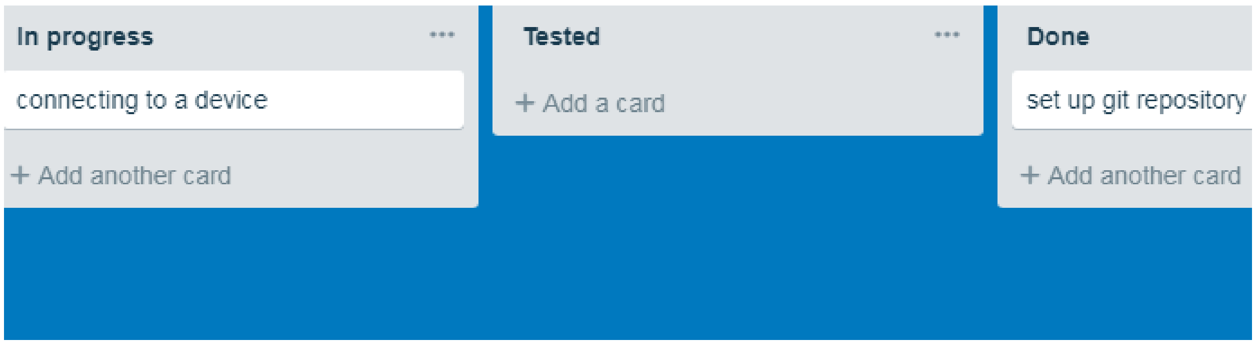


Fig. 3 - Backlog 2

Risk analysis:

Problem	Severity	Impact	Priority	Handling
Database fails	10	10	High	Backup database
External factors that interfere with the development	10	10	High	Reschedule tasks. Set up milestones.
Device breaks	7	7	Medium	Recall device

Table 2 - Risk analysis

Based on the review thus far, the most important tasks at the current time is to have a line of communication with the device. Firstly, the device comes with its own documentation that needs to be reviewed. Based on that, a connection should be made that is capable of receiving and sending messages to the device. Lastly, the message then has to be re-coded as JSON and sent to the backend. Comparing the available time and the tasks described thus far, the estimation is that the workload is sufficient for sprint 1.

Implementation

Sprint 1

The first step is to design a class diagram for the part of the system that interacts with the OBD. The way that the OBD communicates is over tcp. To allow it to communicate a Host class will be created. Every new device that tries to connect to the Host class will instantiate a *Client* class. As soon as the device is connected, any messages that are transmitted have to be processed.

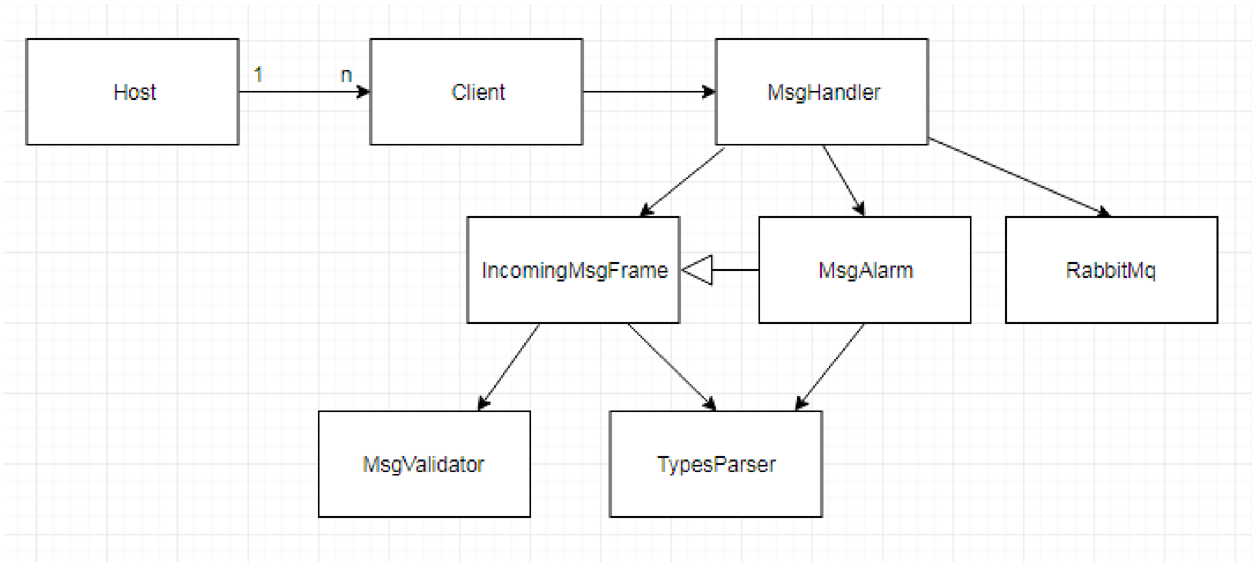


Fig. 4 - Design class diagram

The Client class will handle all inbound and outbound communications with the device it represents. Every message read will go through *MsgHandler*. *MsgHandler* will have a separate *IncomingMsgFrame* class that will contain a general framework of a message. This will validate the integrity of the received message. With that, *MsgHandler* directs the message to the proper class. Since the system only cares for messages of type Alarm, only *MsgAlarm* class will be implemented. *MsgAlarm* will also inherit from *IncomingMsgFrame* since *MsgAlarm* contains a more detailed handling of the message.

After handling the message, the results are returned to the *MsgHandler* that passes them to the RabbitMQ for publishing to a message queue. To help with parsing and validating the *IncomingMsgFrame* and *MsgAlarm* Classes will use *MsgValidator* and *TypesParser* classes.

The interaction diagram

The interaction diagram follows the same classes as the design class diagram but with the emphasis on the lifecycle of the message. The message comes in through the *Client* class. It is picked up by *MsgHandler*. To make sure of the integrity of the message, it is passed through *IncomingMsgFrame* and *MsgValidator*. If the message is validated and is of type alarm, it gets passed to *MsgAlarm* by the *MsgHandler*. The message is then decoded with the use of *TypesParser* and a response is encoded with the use of *MsgValidator*. After that, the *MsgAlarm* returns the decoded message in the form of a JSON and a response for the device in the form of a hex. The *MsgHandler* then sends the JSON to the *RabbitMQ* class and the response to the client class.

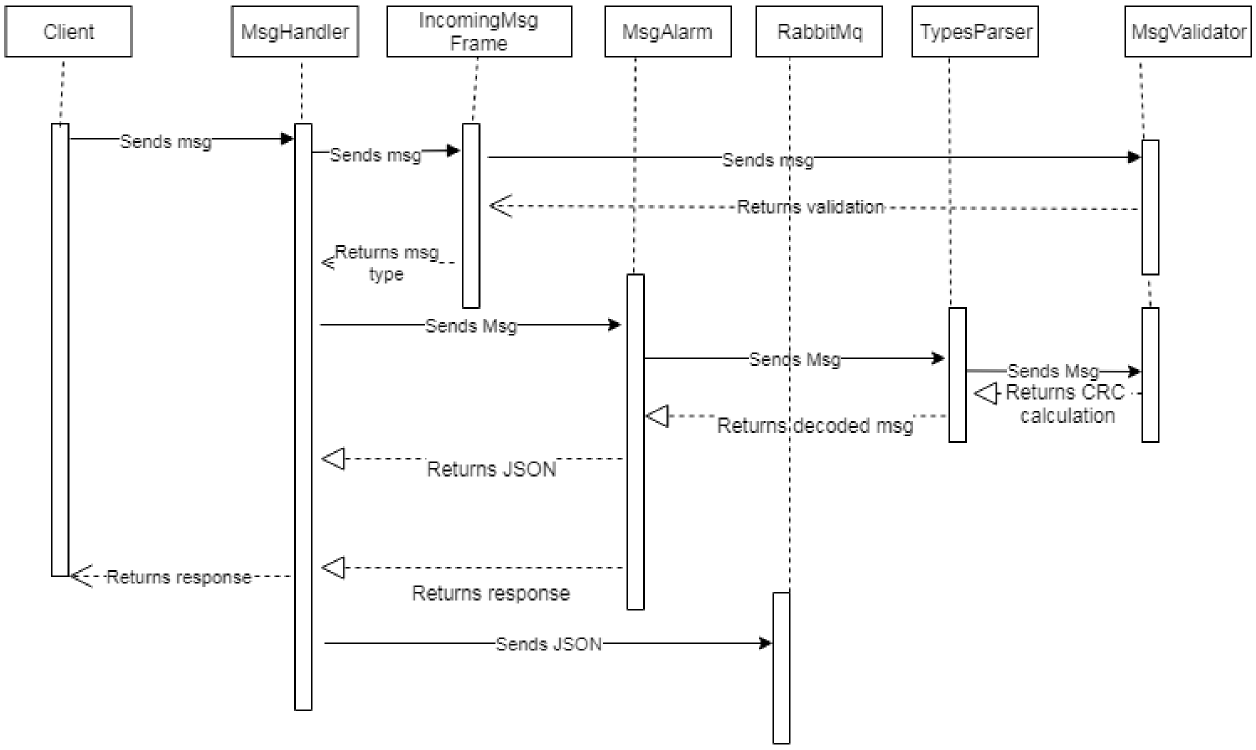


Fig. 5 - Interaction diagram

Device

The OBD can be connected to a car and transmit diagnostics data about the vehicle to a preset ip address over a tcp/ip connection. The transmissions are in hex format. The messages that are being sent and expected to be received all follow one general packet type

Packet header	Packet Length	Device code	Event code	Event data	CRC Sum	Packet Tail
A	B	C	D	E	F	G

Table 3 - Generic packet format

- A. The packet header has a size of 2 Bytes, and is fixed as 4040.
- B. The packet length has a size of 2 Bytes, from header to tail, length range includes A,B,C,D,E,F,G.. Max length cannot be more than 1024 bytes.
- C. The devices personal code has a size of 12 Bytes and a data type of ASCII. If the ASCII device code is shorter than 12 bytes it automatically fills the rest with “\0”
- D. The event code has a size of 2 Bytes. There are multiple types of events that can be sent by the device or by the server.
- E. The event data has an unfixed length as per the specific event code. It may be 0 Bytes.
- F. CRC Sum has a size of 2 Bytes, calculated from header to event code, not including CRC sum or Packet tail, CRC sum algorithm range is A,B,C,D,E.
- G. Packet tail has a size of 2 Bytes, and is fixed as 0d0a.

The CRC -cyclic redundancy check- is a code that verifies the integrity of the data and detects any defects or noise in the transmission. The way they work is by implementing a polynomial division algorithm on the data. The resulting remainder is annexed to the message and sent through the tcp. On the other side the same algorithm should be run again on the same parts of the message. If the remainders are the same then the program knows that the message received is the same message that was sent.

Besides the device code the data types are signed or unsigned integers of different byte sizes, between 1 and 4.

Following the requirements, the program needs to capture and process the parking of the car event. The design specifications of the OBD show that the event is of the type Alarm. This type of event is triggered in many situations. The server has to reply to this event with an acknowledgement event. If there is no acknowledgement, the device will send the message 2 more times, each at 5 second intervals. After the 3 attempts the device will consider the message is successfully transmitted.

Below is a detailed schema of the Alarm event and all of its components. This will be used to decode the message.

Item	Field name	Data type	Length	Description
1	Event code	U16	2	0x2003
2	Random No.	U16	2	UNIT customized random number, used for confirming if platform has replied the event or not.
3	Alarm Tag	U8	1	=0 Alarm end =1 New alarm
4	Alarm No.	U8	1	Alarm No. definition is same with the Parameter No. of Parameter Definition List Alarms with New Alarm and Alarm End (two states): 0x01 MIL on 0x02 Idle Engine 0x03 Fatigue Driving 0x04 Low voltage 0x05 High temperature 0x06 Towing 0x07 Speeding 0x08 High RPM 0x0f ACC ignition on Alarms with only New Alarm state: 0x09 Hard acceleration 0x0a Hard braking 0x0b Quick turn 0x0c Power on **0x0d Power off** 0x0e SOS
5	Alarm Threshold	U16	2	Details regarding thresholds. Does not apply to every type of alarm
6	Alarm value	U16	2	Details regarding values. Does not apply to every type of alarm
7	RTC Time	UTC_TIME.	6	UNIT RTC time when alarm occurs
8	GPS Data	GPS_INFO.	21	UNIT real-timely captured GPS info.

Table 4 - Alarm packet components

The reply the device expects is the event pair and the random no. that it sent. The Alarm number that the server needs to listen for ix 0x0d. That represents that the car's engine is turned off.

Item	Field name	Data type	Length	Description
1	Event code	U16	2	0xA003
2	Random No.	U16	2	Initially sent by the device. Returned by the server.

Table 5 - Alarm response components

Device hex

Below there is an example of a hex string sent by the device. This string represents the type of message the adaptor is expected to receive in the case of an alarm of type *Power off*. Below the hex code is a breakdown of the code and what it represents.

```
4040 3900 3247512d3136303130303838 0320 1000 01 0d 0000 0000
0703110f1b39 0703110f1b03 0c 549e3d0c 2aec2002 0000 0000 0000 3408
0d0a
```

4040	Packet head
3900	Packet length, 57 hex chars
3247512d3136303130303838	Device number (ASCII code) and it is 2GQ-16010088
0320	Event 2003 (little endian mode) type Alarm
1000	Random number provided by the device
01	Alarm tag (00=end, 01=new)
0d	Alarm number, is 13
0000	Alarm threshold, does not apply
0000	Alarm current value, does not apply
0703110f1b39	UTC time, from hex to decimal
0703110f1b03	UTC time (part of the GPS data)
0c	Hex, GPS is not fixed
549e3d0c	North latitude[*]

2aec2002	East longitude[*]
0000	Speed 0
0000	Course 0
0000	Altitude 0
3408	CRC code
0d0a	Packet tail

Table 6 - Hex code components

For the OBD-tcp adaptor to capture the proper alarm event and to send it as a JSON file through the RabbitMQ it first has to create a communication channel with the device. The adaptor first needs to capture the message and identify the device. At the same time the adaptor has to have in place the ability to reply to the alarm event with an acknowledgement event.

To communicate with the device a socket handler has to be implemented. For this, python seemed to be a good fit for it's lightweight and fast processing. The python module will be in charge only of receiving and transmitting data from and to the OBD and then transmitting that data to a RabbitMQ exchange. The module will also be connected to a rabbit queue from which it will be possible to receive commands for the devices. By implementing this architecture, multiple python modules can be ran in parallel based on the traffic and the needs. At the same time multiple other modules can request data by sending a command to one of the queues.

The asyncore.dispatcher works by creating network channels that it adds to a network map. By virtue of a polling loop these channels are constantly listened to in the background.

Asyncore creates a wrapper around socket objects. By wrapping the socket, it allows for higher level calls to be made easier to implement. Some of its methods come with implication that certain events have taken place. A connection is implied when the socket has a read or a write event. A closed connection is implied when the socket has a read event with no data.

By following the example provided for a basic echo server[2], the program implements a server that instantiates an personal socket channel for each connection. Following this implementation each handler will be responsible for its own communication channel

The host server needs a host and a port to accept connections. For this project the host will be localhost and the port open for connections will be 3333. By creating a socket and binding the host to that address, the host will now listen to any events on that socket.

When a read event is picked up by the socket, an attempt at a connection takes place. If it succeeds the return values are a new socket object. This socket will be used to send and receive information from and to the connection. The second value is the address on the other side of the connection. If the attempt fails, the return value is None and the server continues listening for future connections.

For every succeeded socket connection, a new *Client* class-type object is created. The purpose of the client is to keep a channel open between the server and the OBD. The client is in charge of accepting and transmitting messages.

Having received the message, the adaptor has to store and decode the message. By sending the message to a message handler, the message can be redirected towards the proper event. Before that, the message needs to be sanitized and the information about the packet that are inside the packet need to be verified. The frame of the packet: header, length, crc sum and tail all return different error statuses if they fail.

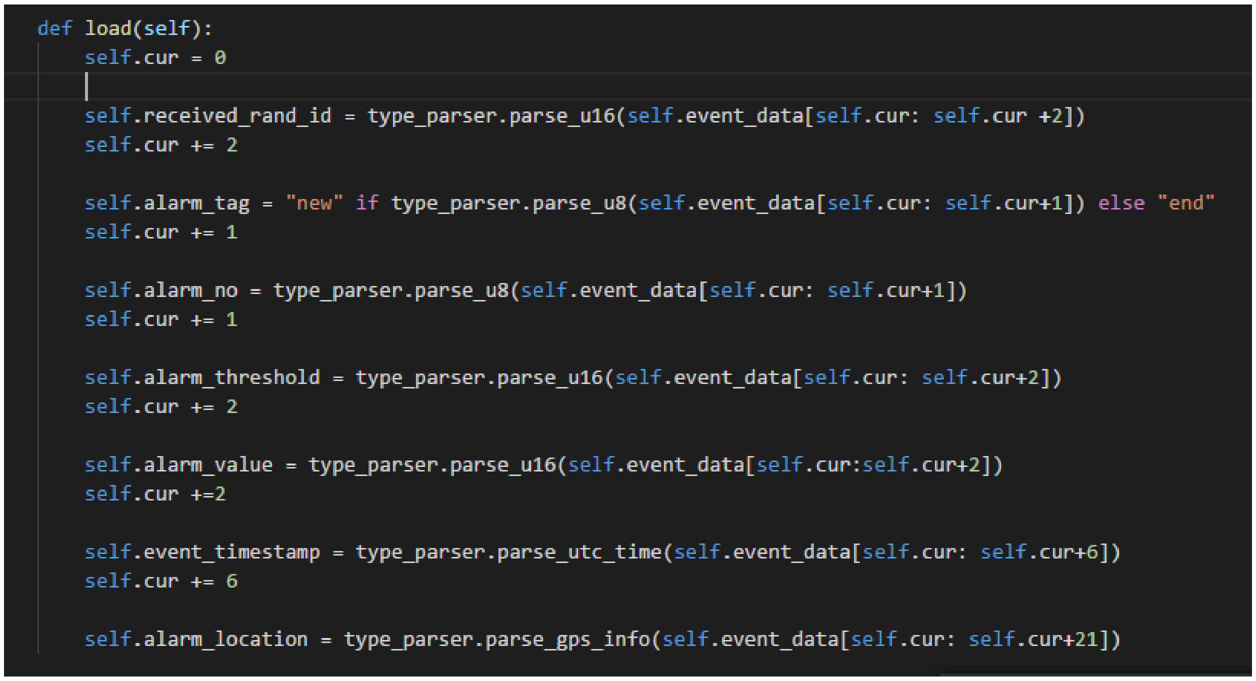
To be able to read and understand the different hex codes the device is transmitting, the program needed to use a conversion module. The struct library[3] has conversions for all data types used by the OBD. At the same time, struct offers both packing and unpacking of data. In the documentation there can be observed the different formatting types and what they return based on size. The “<” symbol refers to the order in which the bytes for one specific value are expected. A table with the different data types used in the program can be seen below.

No	Expected data type	Unpacking function
1	1 Byte unsigned integer	unpack('B', barray)
2	S8 1 Byte signed integer.	unpack(‘b', barray)
3	U16 2 Byte unsigned integers.	unpack('<H', barray)
4	S16 2 Byte signed integers.	unpack('<h', barray)
5	U32 4 Byte unsigned integers.	unpack('<I', barray)
6	S32 4 Byte unsigned integers	unpack('<i', barray)

Table 7 - Hex conversions

Before decoding the message, it has to be sanitized. This comes in four steps.The first step in sanitizing is checking the head by comparing it against the fixed value. The second step is to compare the size of the packet with the stored value. The third step is to check the crc code. For that the program needs to implement the same crc code that is provided by the company and run it on the packet following the documentation. The last step is to check the tail by the same method as the head.

If the message is sanitized it can then be sent to the specific event handler. There the event data can be processed. By using the same parsing functions as described above, together with the documentation of the event, the adaptor can save all the relevant data for the particular type of event.

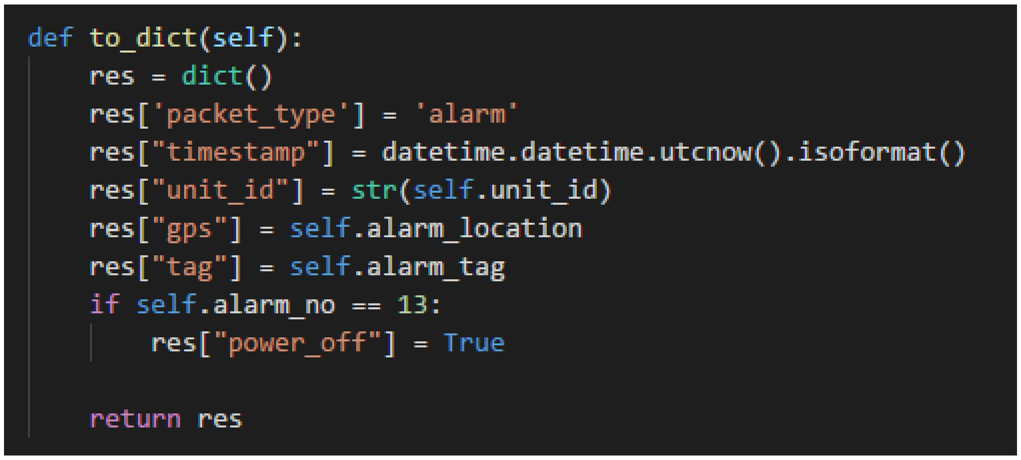


```
def load(self):
    self.cur = 0

    self.received_rand_id = type_parser.parse_u16(self.event_data[self.cur: self.cur +2])
    self.cur += 2

    self.alarm_tag = "new" if type_parser.parse_u8(self.event_data[self.cur: self.cur+1]) else "end"
    self.cur += 1

    self.alarm_no = type_parser.parse_u8(self.event_data[self.cur: self.cur+1])
    self.cur += 1

    self.alarm_threshold = type_parser.parse_u16(self.event_data[self.cur: self.cur+2])
    self.cur += 2

    self.alarm_value = type_parser.parse_u16(self.event_data[self.cur:self.cur+2])
    self.cur +=2

    self.event_timestamp = type_parser.parse_utc_time(self.event_data[self.cur: self.cur+6])
    self.cur += 6

    self.alarm_location = type_parser.parse_gps_info(self.event_data[self.cur: self.cur+21])
```

Fig. 6 - Alarm message decoder

The parameters are the same as described in the event data. In regards to the scope of the project the important messages needed to be transmitted are the location and the device code. A simple way to send the data through the RabbitMQ is to create a dictionary with key-value pairs. The dictionary stores the pairs in the same format as a JSON file. Because of that, the information can be accessed by the backend. The scope of this project is only related with the "ignition off". For that, the *MsgAlarm* Class must identify the type alarm. As seen in the Fig. 6 the hexcode for the ignition off is 0x0d which translates to the integer 13.

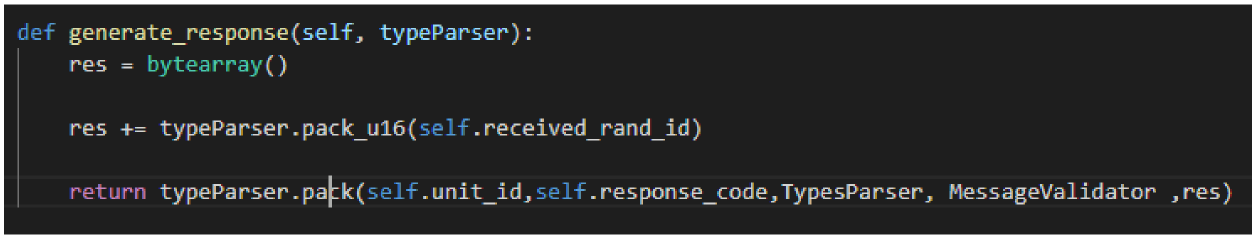
The dictionary will be used for any *alarm* type message. If the system would be fully implemented a separate function would convert the integers to the appropriate messages. However, if the system detects the integer 13, it will add a new parameter to the JSON.



```
def to_dict(self):
    res = dict()
    res['packet_type'] = 'alarm'
    res["timestamp"] = datetime.datetime.utcnow().isoformat()
    res["unit_id"] = str(self.unit_id)
    res["gps"] = self.alarm_location
    res["tag"] = self.alarm_tag
    if self.alarm_no == 13:
        res["power_off"] = True

    return res
```

Fig. 7 - Alarm JSON encoder

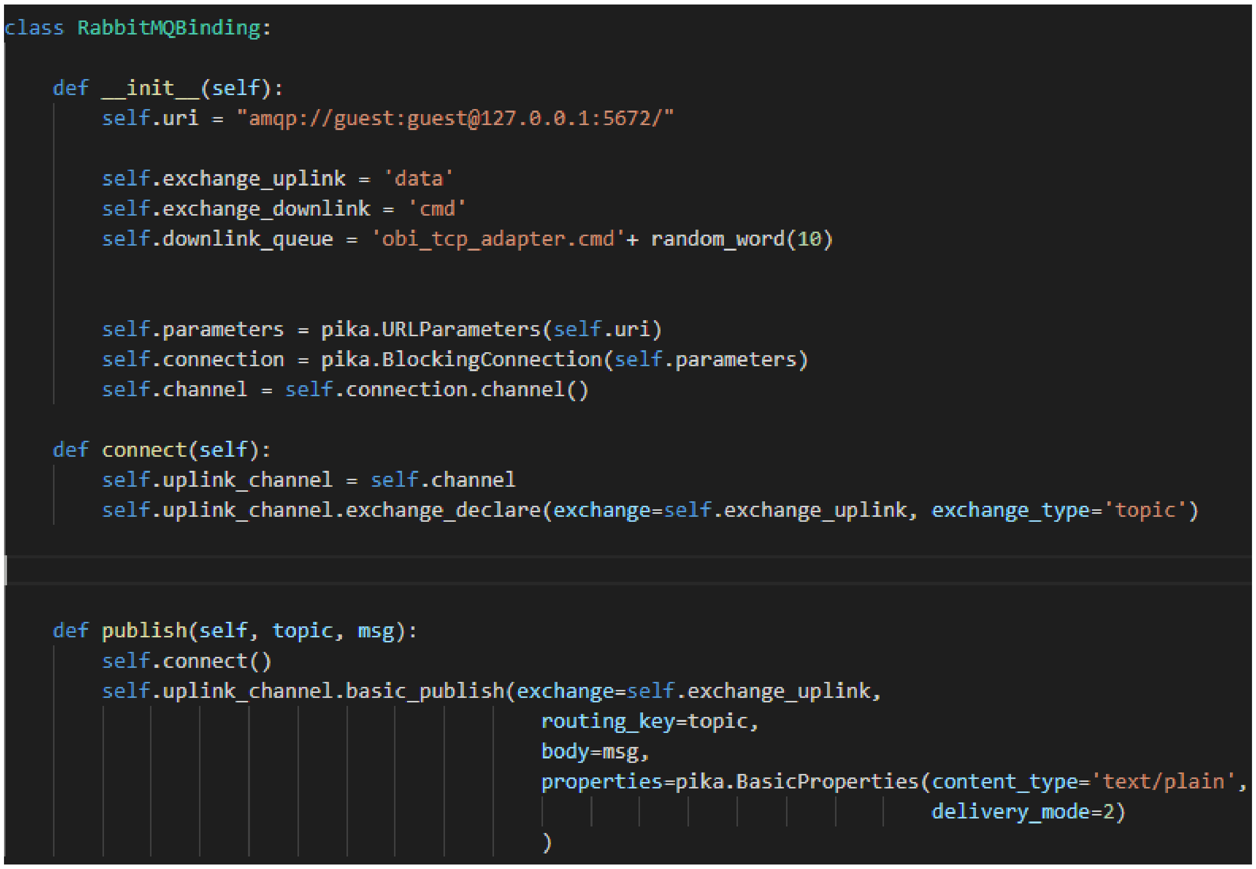
Because the device is waiting for a response, the *MsgAlarm* class will also contain a method to generate the response. This method will contain the random number provided by the device. Together with the device id and the response code for the alarm, it will be packed into a hex string. This hex string will then be sent back to the *Client* class. The *Client* class will then, using asyncore, send the message to the device. The picture below represents the method that creates the reply message.



```
def generate_response(self, typeParser):
    res = bytearray()

    res += typeParser.pack_u16(self.received_rand_id)

    return typeParser.pack(self.unit_id,self.response_code,TypesParser, MessageValidator ,res)
```

Fig. 8 - Alarm response generator

If a JSON has been created by the *MsgAlarm* class, it gets published by the RabbitMQ. When the adaptor is started it connects to an AMQP bus and connects to the upload and download exchanges. The upload exchange is called *data* and it is used to send information from the device to the backend. The download exchange is called *cmd* and it is used to send commands to devices. The adaptor also creates its own queue with unique name, that attaches to the *cmd* exchange, that is done so multiple instances could run at the same time.



```
class RabbitMQBinding:

    def __init__(self):
        self.uri = "amqp://guest:guest@127.0.0.1:5672/"

        self.exchange_uplink = 'data'
        self.exchange_downlink = 'cmd'
        self.downlink_queue = 'obi_tcp_adapter.cmd'+ random_word(10)


        self.parameters = pika.URLParameters(self.uri)
        self.connection = pika.BlockingConnection(self.parameters)
        self.channel = self.connection.channel()

    def connect(self):
        self.uplink_channel = self.channel
        self.uplink_channel.exchange_declare(exchange=self.exchange_uplink, exchange_type='topic')



    def publish(self, topic, msg):
        self.connect()
        self.uplink_channel.basic_publish(exchange=self.exchange_uplink,
                                          routing_key=topic,
                                          body=msg,
                                          properties=pika.BasicProperties(content_type='text/plain',
                                                                          delivery_mode=2)
                                          )
```

Fig. 9 - RabbitMQ class

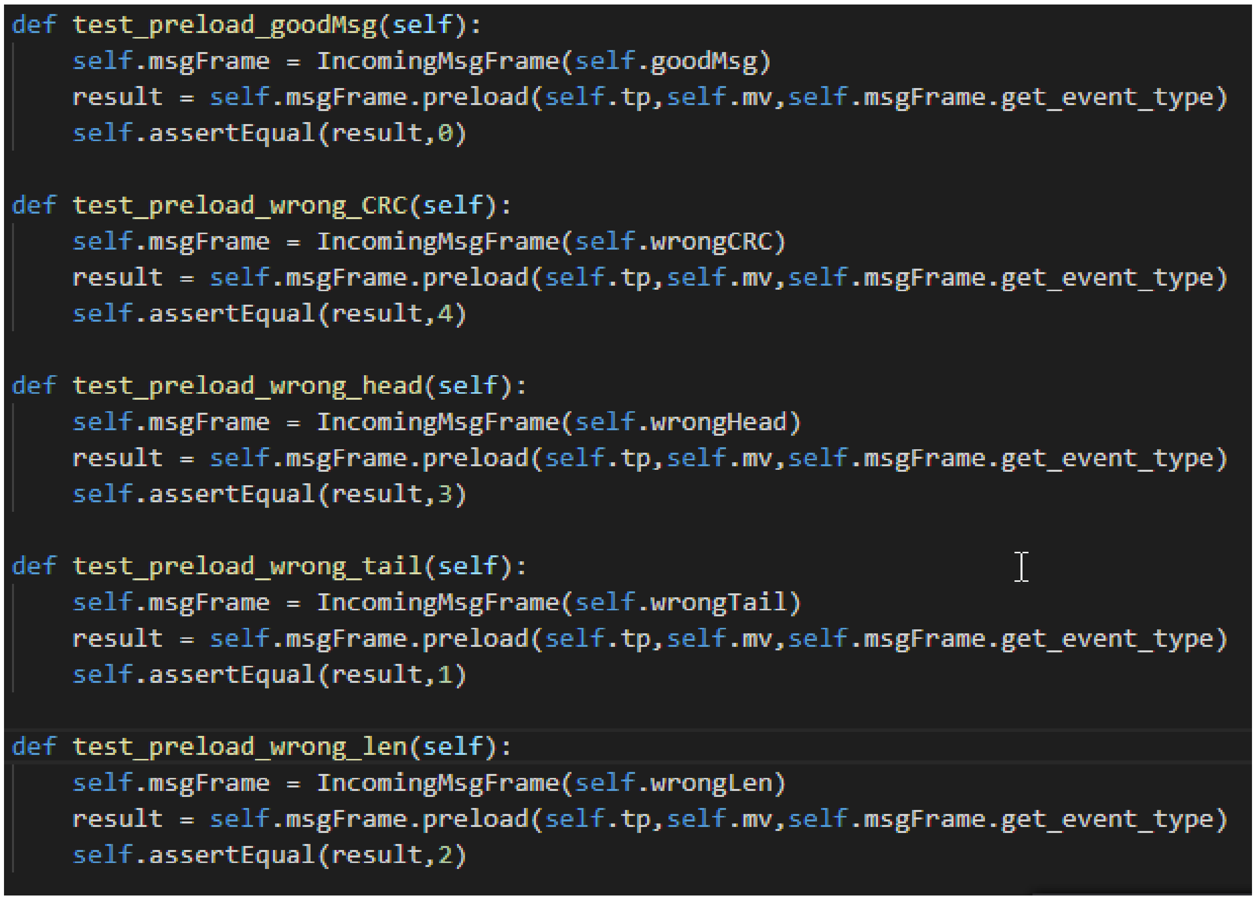
The type of exchange is topic. This will allow messages to be directed via their routing key. The standard message routing key is *<DEVICE_ID>.<EVENT_TYPE>*. With this multiple queues can be set up to listen to specific types of messages.

Below there is an example of what a message will look like. It is based on the previously mentioned hex code.

Exchange	data
Routing Key	2GQ-16010088.alarm
Payload	{"timestamp": "2018-10-06T04:39:39.883000", "tag": "new", "packet_type": "alarm", "unit_id": "2GQ-16010088", "gps": {"speed": 0.0, "fix": false, "course": 0.0, "lng_dir": "E", "lat": 57.04578333333333, "lng": 9.920011666666667, "lat_dir": "N", "altitude": 0.0, "utc_time": "2017-03-07T15:27:03"}}

Table 8 - Rabbit alarm message

Testing



```
def test_preload_goodMsg(self):
    self.msgFrame = IncomingMsgFrame(self.goodMsg)
    result = self.msgFrame.preload(self.tp,self.mv,self.msgFrame.get_event_type)
    self.assertEqual(result,0)

def test_preload_wrong_CRC(self):
    self.msgFrame = IncomingMsgFrame(self.wrongCRC)
    result = self.msgFrame.preload(self.tp,self.mv,self.msgFrame.get_event_type)
    self.assertEqual(result,4)

def test_preload_wrong_head(self):
    self.msgFrame = IncomingMsgFrame(self.wrongHead)
    result = self.msgFrame.preload(self.tp,self.mv,self.msgFrame.get_event_type)
    self.assertEqual(result,3)

def test_preload_wrong_tail(self):
    self.msgFrame = IncomingMsgFrame(self.wrongTail)
    result = self.msgFrame.preload(self.tp,self.mv,self.msgFrame.get_event_type)
    self.assertEqual(result,1)

def test_preload_wrong_len(self):
    self.msgFrame = IncomingMsgFrame(self.wrongLen)
    result = self.msgFrame.preload(self.tp,self.mv,self.msgFrame.get_event_type)
    self.assertEqual(result,2)
```

Fig. 10 - Sanitization method unit tests

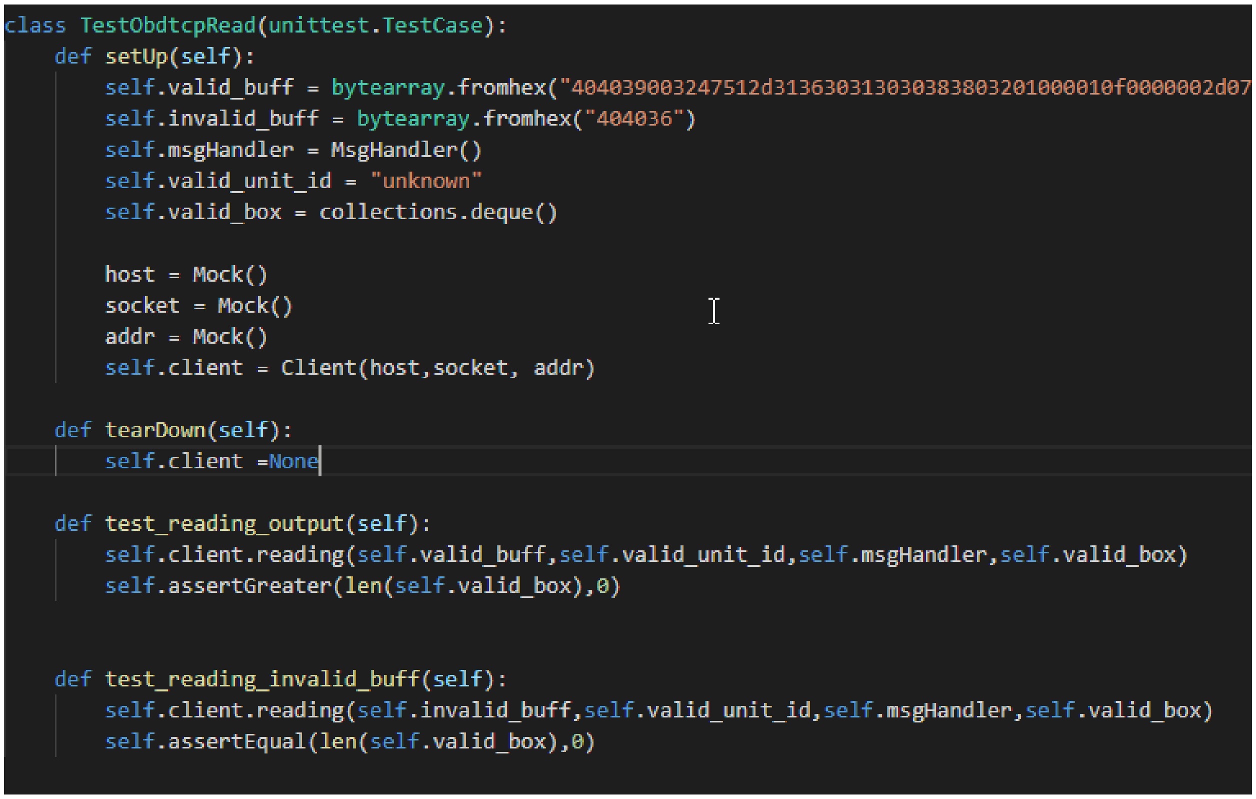
To make sure that the system works as intended, unit tests are written. These unit tests make sure that the program works within specifications. By having unit tests, the system is verified that all the outcomes are handled and work as intended.

The most important part of this sprint is handling and decoding the message. Because of that, the tests will also focus on the different ways the messages that come in can malfunction.

At the end of the sprint all methods should be tested to be considered as done. During the development of the system, refactoring was needed to make some of the methods testable. For a method to be testable, it needed to have all the variables passed in the method's signature. Since some of the methods called other methods, that presented a challenge as to creating the right method signature.

A way to solve this problem was presented by the functional programming that python allows. In python, methods can be passed as variables inside other method signatures, thus rendering them testable.

To make sure a method is properly tested, first a setup of the environment is needed. This is done to make sure only one thing is tested at a time.



```python
class TestObdtcpRead(unittest.TestCase):
    def setUp(self):
        self.valid_buff = bytearray.fromhex("404039003247512d31363031303038383032010000010f0000002d07
        self.invalid_buff = bytearray.fromhex("404036")
        self.msgHandler = MsgHandler()
        self.valid_unit_id = "unknown"
        self.valid_box = collections.deque()

        host = Mock()
        socket = Mock()
        addr = Mock()
        self.client = Client(host,socket, addr)

    def tearDown(self):
        self.client =None

    def test_reading_output(self):
        self.client.reading(self.valid_buff,self.valid_unit_id,self.msgHandler,self.valid_box)
        self.assertGreater(len(self.valid_box),0)


    def test_reading_invalid_buff(self):
        self.client.reading(self.invalid_buff,self.valid_unit_id,self.msgHandler,self.valid_box)
        self.assertEqual(len(self.valid_box),0)
```

Fig. 11 - Mock example

As shown in Fig. 10 all the variables that are needed for the proper execution of the method that is tested are declared in the setup. Because the client requires a Host to attach to, but it is out of the scope of the test, mocks are used. "An object under test may have dependencies on other (complex) objects. To isolate the behavior of the object you want to replace the other objects by mocks that simulate the behavior of the real objects."[4]

To make sure that the tests are independent of each other, after each test a cleanup is run. This is done to make sure that data is not transferred between tests. In the example provided, for each test a new client was created and after each test, the client was destroyed.

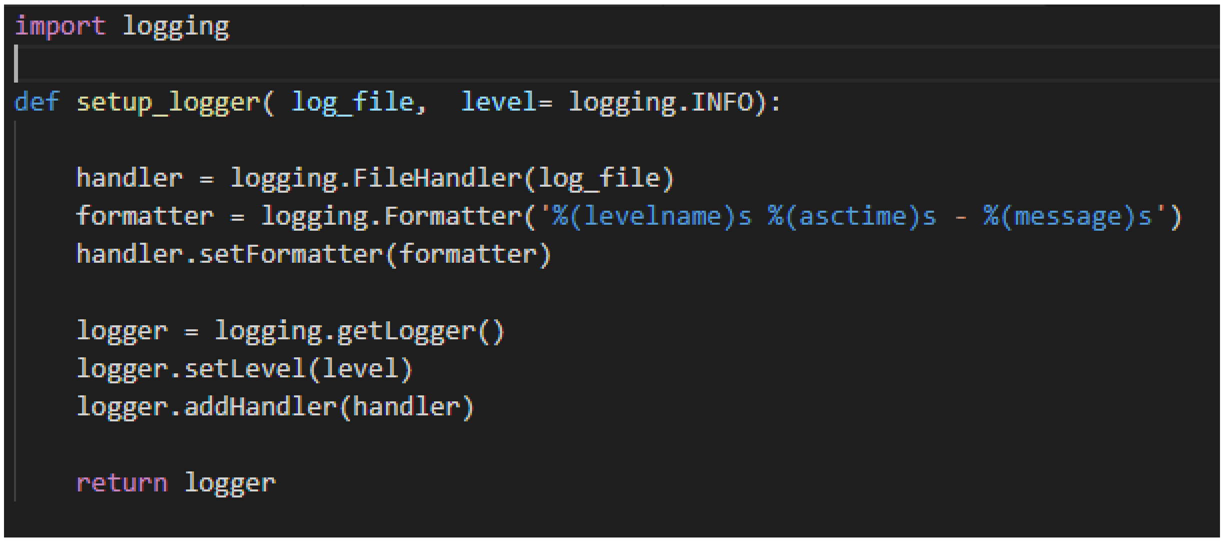
Following the AAA pattern [5], the Arrange part of the unit test is in the setup. The Act is invoked inside each of the unit tests. In each unit test, a different configuration is ran. Based on the expected outcome, each unit test is then asserted. If the result does not match the expected value, the test fails.

If by running the tests and any inconsistencies are discovered, if they can not be quickly fixed, the task will then be reallocated for the next sprint.

Logging

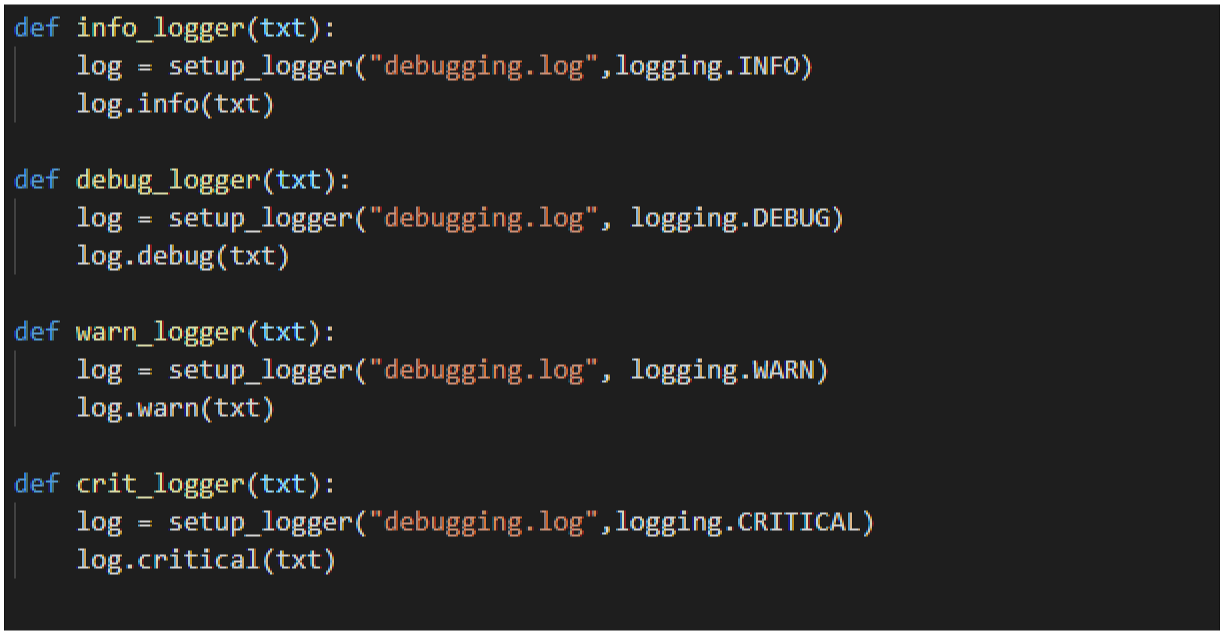
To ensure that the system functions properly even after deployment, a logging system was integrated. With logging, developers and system administrators can identify and pinpoint potential failures. Logging can be used to make sure the system works within normal parameters. It can also be used for better customer support and maintenance.

As shown in the Fig. 11 below, a setup of a logger was implemented. By having this setup, multiple loggers can be instantiated, each with its own log file.
The *FileHandler* comes with the logging module. It is an instance of a *StreamHandler*. It makes sure there is a file to write to and deals with encoding and delays.
The logger has different levels. Based on those levels, non-critical messages can be left out of the file.



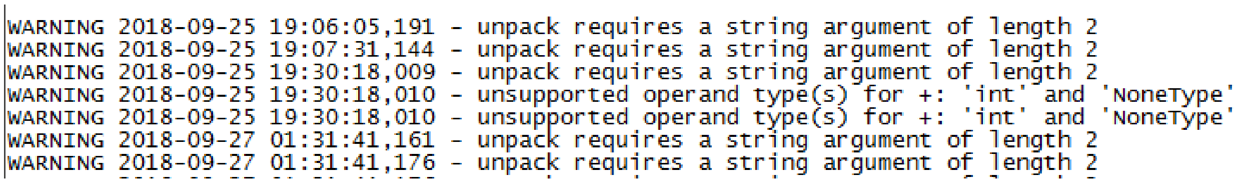
```python
import logging

def setup_logger( log_file,  level= logging.INFO):

    handler = logging.FileHandler(log_file)
    formatter = logging.Formatter('%(levelname)s %(asctime)s - %(message)s')
    handler.setFormatter(formatter)

    logger = logging.getLogger()
    logger.setLevel(level)
    logger.addHandler(handler)

    return logger
```

Fig. 12 - Logger setup



```python
def info_logger(txt):
    log = setup_logger("debugging.log",logging.INFO)
    log.info(txt)

def debug_logger(txt):
    log = setup_logger("debugging.log", logging.DEBUG)
    log.debug(txt)

def warn_logger(txt):
    log = setup_logger("debugging.log", logging.WARN)
    log.warn(txt)

def crit_logger(txt):
    log = setup_logger("debugging.log",logging.CRITICAL)
    log.critical(txt)
```

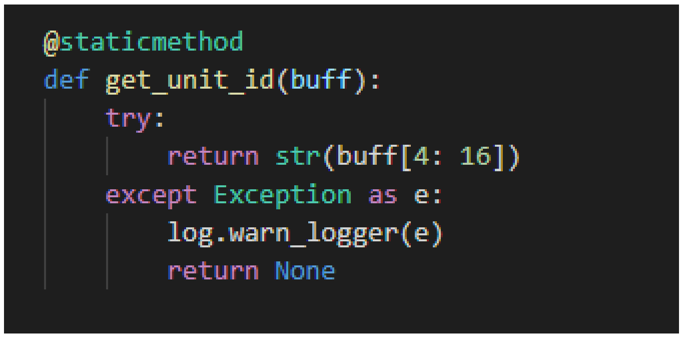
Fig. 13 - Functions for logging



```
WARNING 2018-09-25 19:06:05,191 - unpack requires a string argument of length 2
WARNING 2018-09-25 19:07:31,144 - unpack requires a string argument of length 2
WARNING 2018-09-25 19:30:18,009 - unpack requires a string argument of length 2
WARNING 2018-09-25 19:30:18,010 - unsupported operand type(s) for +: 'int' and 'NoneType'
WARNING 2018-09-25 19:30:18,010 - unsupported operand type(s) for +: 'int' and 'NoneType'
WARNING 2018-09-27 01:31:41,161 - unpack requires a string argument of length 2
WARNING 2018-09-27 01:31:41,176 - unpack requires a string argument of length 2
```

Fig. 14 - Logging file output

If the file does not exist, it is created by the *FileHandler*. New logs are added to the file based on the described format type. To write to a log file, the logger file is imported. This gives the program access to the logging methods that can be created based on the needs of the programmer. In the Fig. 13 below, an example of such a method is called on the event of an exception.



```python
@staticmethod
def get_unit_id(buff):
    try:
        return str(buff[4: 16])
    except Exception as e:
        log.warn_logger(e)
        return None
```

Fig. 15 - Logger usage example

At the end of the first sprint a review is held to make sure all the tasks were completed in the allocated time and if there were any setbacks and how they were dealt with. A big part of the sprint was allocated to understanding the system and the device. For this, diagrams were made and the device documentation was reviewed. The implementation ran according to plan but during the testing phase, some unforeseen refactoring was needed. This took up some of the time from the testing phase and because of that only the critical parts of the system were tested.

Sprint 2

Based on the tasks completed in the last sprint, it was decided that the main focus of this sprint would be to implement the backend. For this sprint, the system should implement a RabbitMQ receiver and two databases. If the message comes from a device that is located in a marked parking lot, the device's owner will receive a message. For this, further research is needed to make sure the components communicate with each other in an established manner.

For the backend it was decided that an MVC solution will provide the necessary components that are required for this system. Based on the experience of the developer, C# is the recommended language for this implementation. It provides a logging in system with authentication and authorization. This will be useful in dealing with users, it will provide adequate protection and it is easy to implement. Because of the popularity, it also provides packets for services such as RabbitMQ and also other technologies such as websockets, that can be used to push the data to the user.

An MVC solution is adopted because it provides a package that comes with a well-established architecture, separation of concerns and a web solution that works on multiple devices.

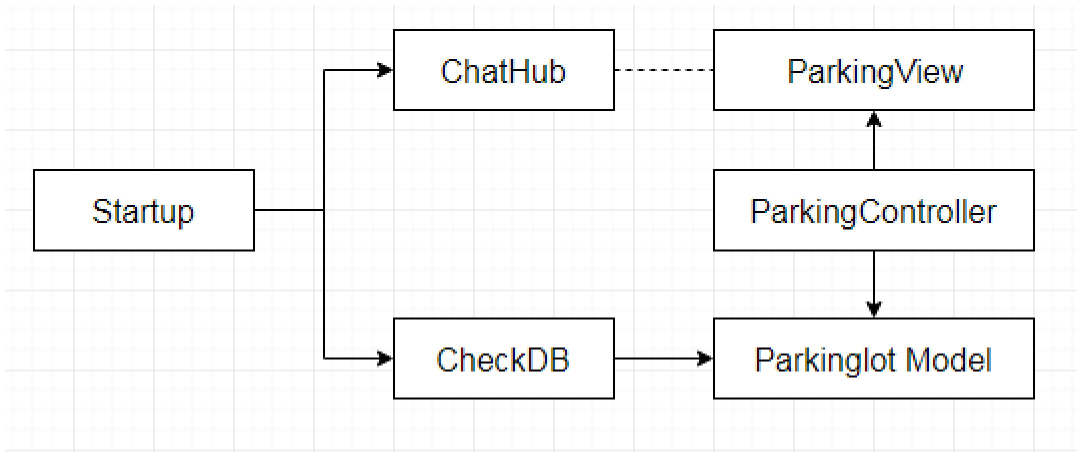


Fig. 16 - Backend architecture

By implementing an MVC solution, part of the architecture is already established. What is still required is to implement a RabbitMQ receiver. That function will reside in the startup class. When triggered by an incoming message, it will prompt the CheckDB class to find if there is a matching entry and identify the user based on the device id.

Because the message is pushed from the device to the user, the system requires a way to notify the user. For this, SignalR provides a viable solution. "ASP.NET SignalR is a new library for ASP.NET developers that makes developing real-time web functionality easy. SignalR allows bi-directional communication between server and client. Servers can now push content to connected clients instantly as it becomes available. SignalR supports Websockets, and falls back to other compatible techniques for older browsers” [6]

Using the standard MVC architecture, the three layers are created. The view uses SignalR javascript that allows for a backend-frontend connection. Using this, messages can be user targeted.

The interaction diagram

The interaction diagram follows the same classes as the design class diagram but with the emphasis on the lifecycle of the message. The message is received by the *Startup* class. If the message is of type Alarm and contains the parameter for the engine turn off, it will send the device id and location to the *CheckDB* class. The *CheckDB* will check the information and if valid will return the username and the parking lot. This information will then be sent to the user by using the *ChatHub*.

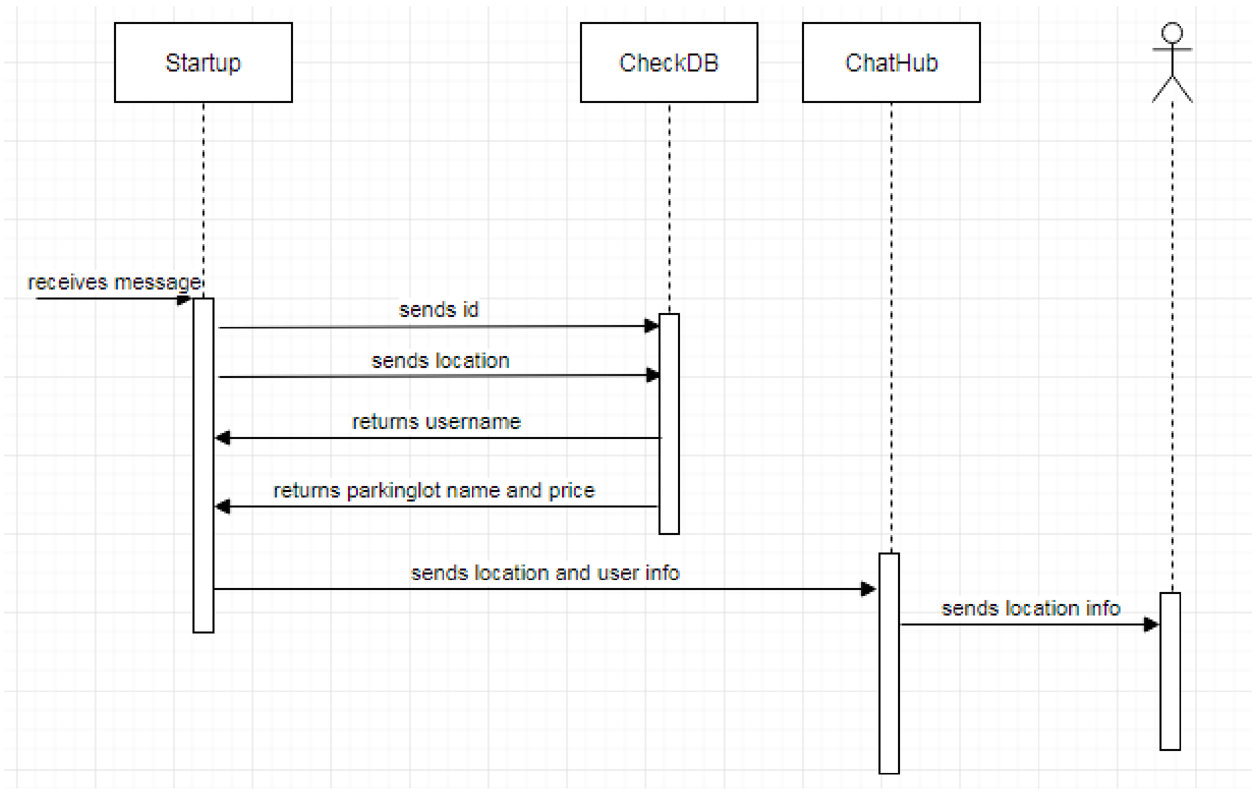
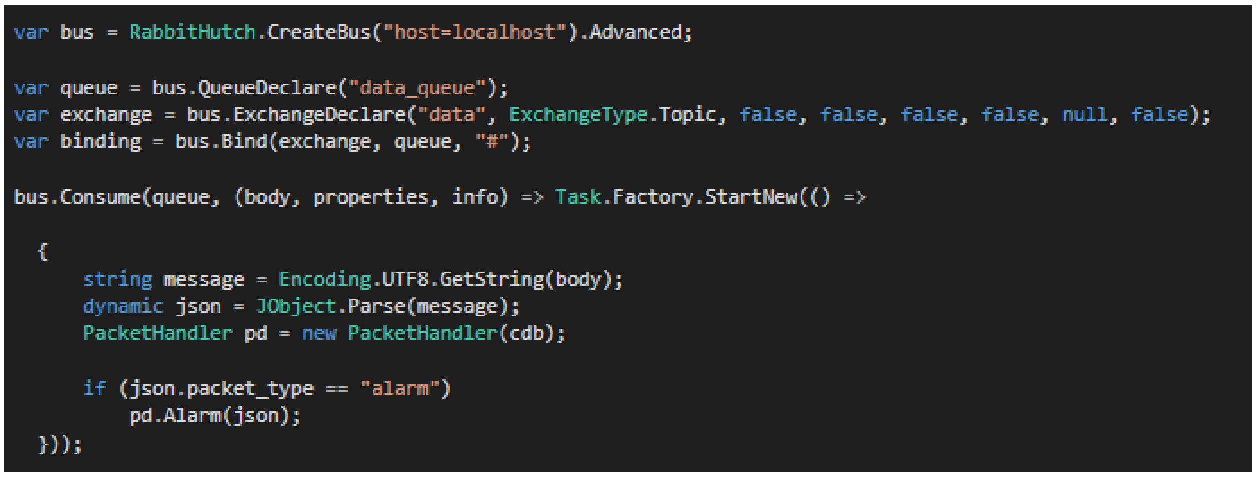


Fig. 17 - Backend interaction diagram

RabbitMQ

To receive the message, a secondary implementation of RabbitMQ is needed to function as a receiver. This receiver will connect to the same exchange uplink that was present in the python implementation. To this exchange, it will add a queue that it will use to consume incoming messages. For now, the binding key for this queue will allow all messages to pass through. The only messages that are being sent are of type alarm but dedicated queues can always be created based on the traffic. These queues can then have binding keys based on the event type.

Once a message comes through the queue, a new task is created. This task will execute the rest of the algorithm asynchronously, thus allowing the program to continue working. Having the alarm message template, the message can then be identified and passed to the appropriate method.



```
var bus = RabbitHutch.CreateBus("host=localhost").Advanced;

var queue = bus.QueueDeclare("data_queue");
var exchange = bus.ExchangeDeclare("data", ExchangeType.Topic, false, false, false, false, null, false);
var binding = bus.Bind(exchange, queue, "#");

bus.Consume(queue, (body, properties, info) => Task.Factory.StartNew(() =>

  {
      string message = Encoding.UTF8.GetString(body);
      dynamic json = JObject.Parse(message);
      PacketHandler pd = new PacketHandler(cdb);

      if (json.packet_type == "alarm")
          pd.Alarm(json);
  }));
```

Fig. 18 - RabbitMQ receiver

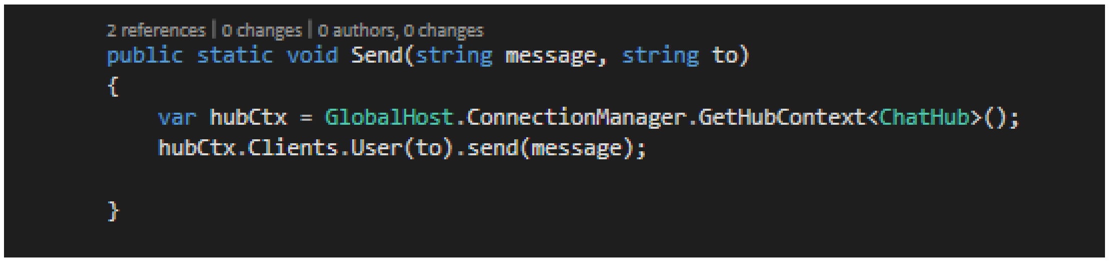
SignalR

To be able to send information from the server to the user, a new technology must be introduced. For this project, SignalR was used. It allows for easy communication in real time between the user and the server. To achieve that, it uses websockets to create a channel of communication between the user and the server. This channel can then be accessed by both parties. If either the server or the client are not compatible with websockets, SignalR will switch to an older technology. The most basic technology is long polling. It mimics real time by having the client asynchronously send requests to the server.

Another reason to use SignalR is that any updates to the underlying technologies do not affect the SignalR implementation. It is also cross-platform compatible. The client code is written in JavaScript, which allows it to run on any device.

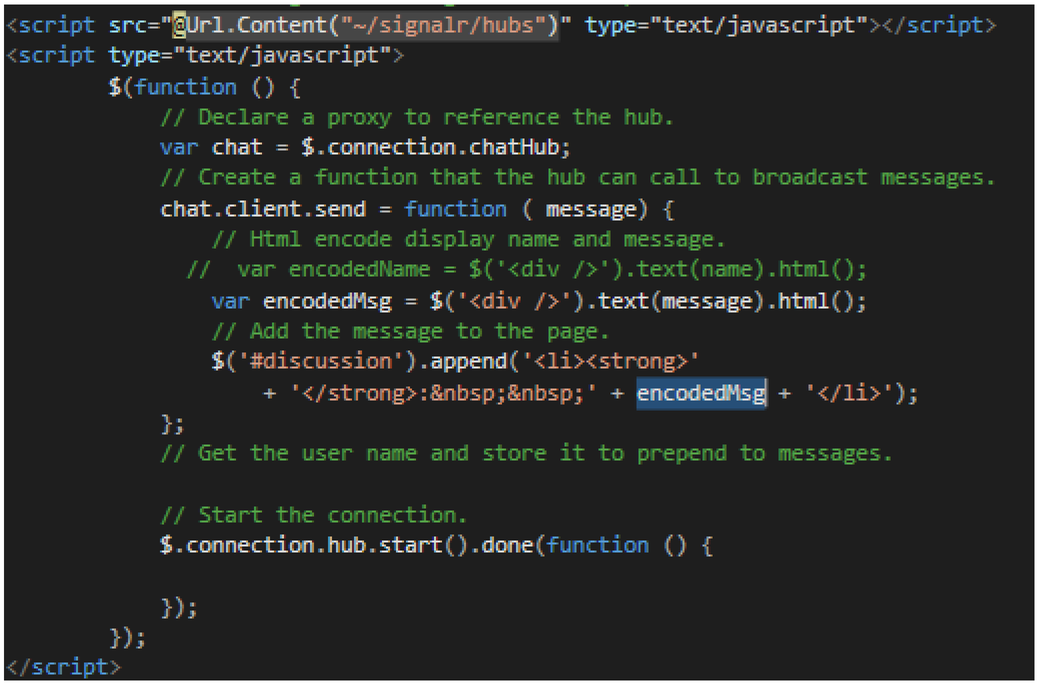
Each time there is a new client-server connection it is given an id and assigned to a SignalR map. To make a SignalR connection both the server and the client have to have an implementation of the code.

Below there is an example of how this implementation is done. The server receives a message and an user. It tracks the SignalR session based on the user and tries to call the *client.send()* function. This way, only that specific user sees the information.



```
2 references | 0 changes | 0 authors, 0 changes
public static void Send(string message, string to)
{
    var hubCtx = GlobalHost.ConnectionManager.GetHubContext<ChatHub>();
    hubCtx.Clients.User(to).send(message);

}
```

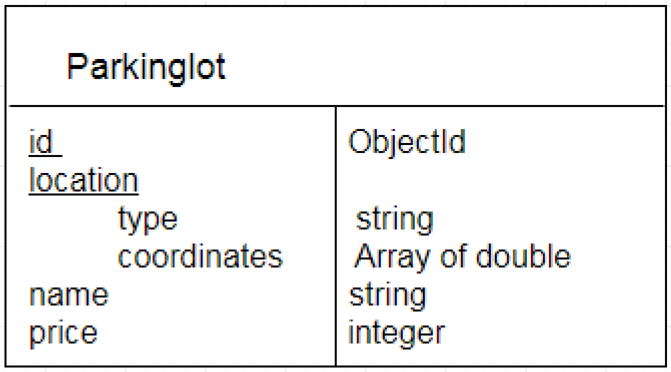
Fig. 19 - server side SignalR implementation



```
<script src="@Url.Content("~/signalr/hubs")" type="text/javascript"></script>
<script type="text/javascript">
        $(function () {
            // Declare a proxy to reference the hub.
            var chat = $.connection.chatHub;
            // Create a function that the hub can call to broadcast messages.
            chat.client.send = function ( message) {
                // Html encode display name and message.
              //  var encodedName = $('<div />').text(name).html();
                var encodedMsg = $('<div />').text(message).html();
                // Add the message to the page.
                $('#discussion').append('<li><strong>'
                    + '</strong>:&nbsp;&nbsp;' + encodedMsg + '</li>');
            };
            // Get the user name and store it to prepend to messages.

            // Start the connection.
            $.connection.hub.start().done(function () {

            });
        });
</script>
```

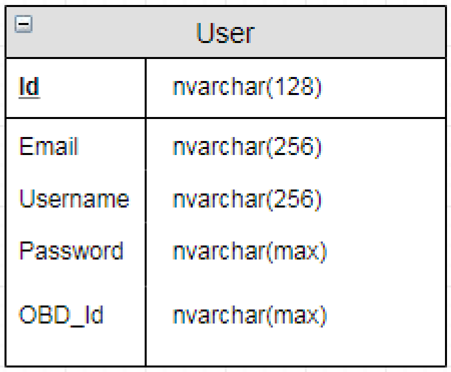
Fig. 20 - Client side SignalR implementation

Databases



Parkinglot	
id	ObjectId
location	
type	string
coordinates	Array of double
name	string
price	integer

Fig. 16 - NoSQL DB



User	
Id	nvarchar(128)
Email	nvarchar(256)
Username	nvarchar(256)
Password	nvarchar(max)
OBD_Id	nvarchar(max)

Fig. 17 - SQL DB

In order for the system to work, it needs to store information about its users and parking lots. For the system to be optimized, it is recommended that the two tables be in different databases.

NoSQL

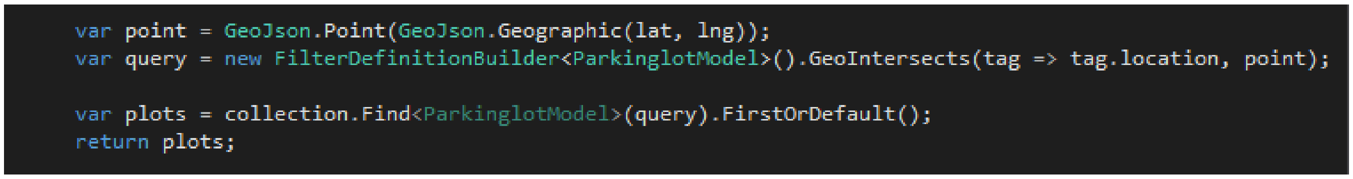
By placing the location in a NoSQL database it will allow for faster queries. NoSQL allows for an unspecified number of geolocation points on each row. At the same time, this database will change very rarely; therefore copying or caching the database will increase performance. To implement a NoSQL database, Mongo was chosen. Mongo is widely used NoSQL database that is free to use and allows for indexing. Mongo also allows for geographical locations to be indexed and queried.

Mongo supports GeoJSON objects. For a GeoJSON to be valid, the object must contain a key-value pairs of the type, and a key-value pair of coordinates. These need to be passed as an array of latitude and longitude coordinate pairs. To allow polygon related searches, the first coordinate pair needs to be the last coordinate pair. Additional information regarding parking lots is required, such as the name and the cost per hour. These can be added as additional key-value pairs.

JSON example of a Parkinglot object:

```
{"Location":{     "type”: “Polygon",
                  "coordinates":[[57.048422, 9.926281],
                  [57.047891, 9.927762],
                  [57.047442, 9.927193],
                  [57.047643, 9.925793],
                  [57.048422, 9.926281]]
                  },
"name": "Friis parking lot",
"Price": 50}
```

The query that needs to be performed on this database is to return an object if the coordinate set provided is inside a polygon. First, a *GeoJson* point must be created based on the provided arguments. Secondly, using the mongo library for C#, a query is created that checks if the point intersects with any of the stored locations. Applying that query on the *Parkinglot* table will return a *ParkinglotModel* object. The *ParkinglotModel* is based off of the *Parkinglot* JSON and contains the same information.



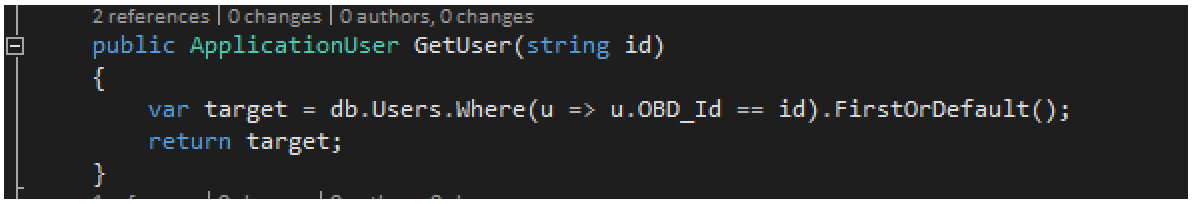
```
var point = GeoJson.Point(GeoJson.Geographic(lat, lng));
var query = new FilterDefinitionBuilder<ParkinglotModel>().GeoIntersects(tag => tag.location, point);

var plots = collection.Find<ParkinglotModel>(query).FirstOrDefault();
return plots;
```

Fig. 21 - NoSQL Geolocation query

SQL

The user's information is stored in a separate database. The main reason for this is that the user information is used in other parts of the system. By using MVC identity, it also provides security in the form of authentication and authorization.

To assist with the implementation and usage of the SQL database, a code first approach of entity framework was used. By creating models, entity framework can use them as a blueprint for the database. One of the changes done to the *AspNetUser* was to add the *OBD_Id*. By adding it to the *ApplicationUser* as a field, the database will be updated. This is done by using the database migration feature. The *OBD_Id* is associated with the user and will be used to find the user based on their device id.



```
2 references | 0 changes | 0 authors, 0 changes
public ApplicationUser GetUser(string id)
{
    var target = db.Users.Where(u => u.OBD_Id == id).FirstOrDefault();
    return target;
}
```

Fig. 22 - retreiving user based on device id

During the second sprint, all the tasks were completed. A lot of time was used on finding third party technologies that could help implement the desired functionality. One such example is SignalR, which allowed the message to be transmitted to the client in real time. There were also two databases implemented one of each is mongoDB. It had a special implementation of geolocation objects that were useful for the project. Because of this, some time was spent studying the mongoDB documentation.

Sprint 3

Based on a meeting with the supervisor, a new task was added, which would calculate the price of parking based on timestamps. This will allow users to know how much they have to pay and will allow for future implementations of third-party payment systems. This task however, changes the entire logic of the system. Because of this, the focus of this sprint will only be this task.

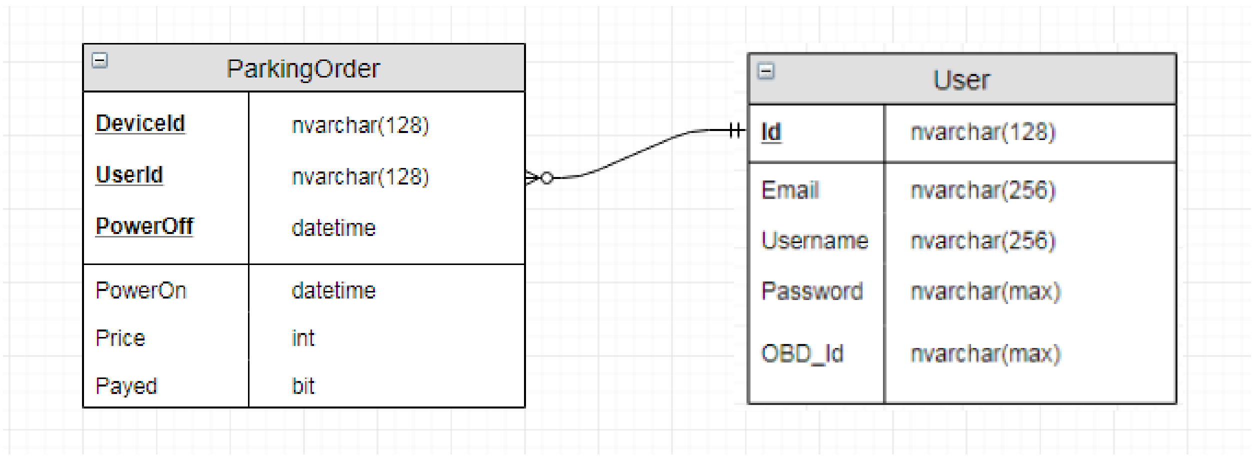


Fig. 23 - parkind information table

Based on a supervisor meeting it was decided to calculate the payment based on the time spent in parking lot. To implement this, a separate table has to be created. This table will be connected to the user based on the user id. This will make the user id a foreign key. The table will also contain device id, timestamps for the engine activity, price and a status of payment. The primary key of the table is composed of *DeviceId*, *userid* and *PowerOff*. This way the table can contain a unique row based on the time the devices were triggered.

When the device sends a *PowerOff* event to the backend, it will check the validity of the information. If the information is valid a new *ParkingOrder* row will be created. This row will store the *deviceId*, *userId*, *price* of the parking lot and the *timestamp*.

When the device sends a *PowerOn* event, the timestamp is added to the row. In addition to this, the total price is calculated and sent to the user. If the timestamp is less than 5 minutes, the timestamp is set to 0. To ensure that the *PowerOff* will be paired with a *PowerOn*, the location check is not executed.

Due to time constraints, this sprint was shorter. However the main task, that was payment calculation, was completed. This will allow integration with different payment options.

Future implementations

Some functionality that would be useful in creating a robust application were:

- External payment: Implementation of third-party paying services.
- Location refinement: To use the phone to more accurately locate the user and to give the user the possibility to confirm.
- Multiple devices per user: This implementation would be useful for company fleets.
- Code optimization: This would include more testing and additional performance tweaks.
- Parking lot integration and CRUD: To automatically add parking lots to the database.
- Dashboards: To allow both the users and the admins quick access to information
- Database backup and cleaning: To have caches of databases regularly updated and cleaned.
- Receipts: to implement an integrated service that would send the receipt to the users after they paid.

Conclusion

During this project, there were many challenges. Having multiple components that all had to integrate into a functional, secure and reliable system takes a lot of time and preparation. The project was completed using the Scrum model. The sprints were completed as expected, with a few setbacks. Talking with the supervisor helped provide a more clear understanding of the system's expectations. This brought some additional functionality that were successfully added to the sprints and later integrated.

While building this project, a better understanding of python was acquired. Having to develop based off of a device helped with a deeper appreciation for proper documentation, as well as the ability to manipulate hex codes. In developing a cross-platform system, multiple systems had to be matched and integrated. For this, time was dedicated to research the best approach. One such example, from the research gathered, would be the existence of mongoDB geospatial queries.

Overall, the project deemed a success and with the addition of a third-party payment system it can be published. Thanks to the project, a lot of knowledge was gained into the workings of low level programming and the integration with other systems.

References

[1] Software Engineering - Ian Sommerville, tenth edition
[2] https://docs.python.org/2/library/asyncore.html
[3] https://docs.python.org/2/library/struct.html
[4] https://stackoverflow.com/questions/2665812/what-is-mocking
[5] https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2017
[6] https://www.asp.net/SignalR
[7] https://docs.mongodb.com/manual/geospatial-queries/#geospatial-data
[8] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

Anex:

CRC16 Checksum Algorithm

```
//CRC Table
const unsigned short crc_table[256]={
0x0000,0x9705,0x2E01,0xB904,0x5C02,0xCB07,0x7203,0xE506,0xB804,0x2F01,0x9605,0x0100,0xE4
06,0x7303,0xCA07,0x5D02,
0x7003,0xE706,0x5E02,0xC907,0x2C01,0xBB04,0x0200,0x9505,0xC807,0x5F02,0xE606,0x7103,0x94
05,0x0300,0xBA04,0x2D01,
0xE006,0x7703,0xCE07,0x5902,0xBC04,0x2B01,0x9205,0x0500,0x5802,0xCF07,0x7603,0xE106,0x04
00,0x9305,0x2A01,0xBD04,
0x9005,0x0700,0xBE04,0x2901,0xCC07,0x5B02,0xE206,0x7503,0x2801,0xBF04,0x0600,0x9105,0x74
03,0xE306,0x5A02,0xCD07,
0xC007,0x5702,0xEE06,0x7903,0x9C05,0x0B00,0xB204,0x2501,0x7803,0xEF06,0x5602,0xC107,0x24
01,0xB304,0x0A00,0x9D05,
0xB004,0x2701,0x9E05,0x0900,0xEC06,0x7B03,0xC207,0x5502,0x0800,0x9F05,0x2601,0xB104,0x54
02,0xC307,0x7A03,0xED06,
0x2001,0xB704,0x0E00,0x9905,0x7C03,0xEB06,0x5202,0xC507,0x9805,0x0F00,0xB604,0x2101,0xC4
07,0x5302,0xEA06,0x7D03,
0x5002,0xC707,0x7E03,0xE906,0x0C00,0x9B05,0x2201,0xB504,0xE806,0x7F03,0xC607,0x5102,0xB4
04,0x2301,0x9A05,0x0D00,
0x8005,0x1700,0xAE04,0x3901,0xDC07,0x4B02,0xF206,0x6503,0x3801,0xAF04,0x1600,0x8105,0x64
03,0xF306,0x4A02,0xDD07,
0xF006,0x6703,0xDE07,0x4902,0xAC04,0x3B01,0x8205,0x1500,0x4802,0xDF07,0x6603,0xF106,0x14
00,0x8305,0x3A01,0xAD04,
0x6003,0xF706,0x4E02,0xD907,0x3C01,0xAB04,0x1200,0x8505,0xD807,0x4F02,0xF606,0x6103,0x84
05,0x1300,0xAA04,0x3D01,
0x1000,0x8705,0x3E01,0xA904,0x4C02,0xDB07,0x6203,0xF506,0xA804,0x3F01,0x8605,0x1100,0xF4
06,0x6303,0xDA07,0x4D02,
0x4002,0xD707,0x6E03,0xF906,0x1C00,0x8B05,0x3201,0xA504,0xF806,0x6F03,0xD607,0x4102,0xA4
04,0x3301,0x8A05,0x1D00,
0x3001,0xA704,0x1E00,0x8905,0x6C03,0xFB06,0x4202,0xD507,0x8805,0x1F00,0xA604,0x3101,0xD4
07,0x4302,0xFA06,0x6D03,
0xA004,0x3701,0x8E05,0x1900,0xFC06,0x6B03,0xD207,0x4502,0x1800,0x8F05,0x3601,0xA104,0x44
02,0xD307,0x6A03,0xFD06,
0xD007,0x4702,0xFE06,0x6903,0x8C05,0x1B00,0xA204,0x3501,0x6803,0xFF06,0x4602,0xD107,0x34
01,0xA304,0x1A00,0x8D05
};
```

```
unsigned short start_crc = 0xffff;
/*
**Calculate crc Checksum buffer.
** Calculate large files CRC Checksum, crc16 function is processing a buffer.
** If a file is relatively large, obviously it can not be read directly into memory,
** So can only read the file segments out for crc check,
** Then circling by passing last crc Checksum to new buffer check function.
** Finally the generated crc Checksum is the file's crc Checksum.
*/
unsigned short crc16(unsigned short crc,unsigned char *buffer, unsigned int size)
{
unsigned int i;
for (i = 0; i < size; i++) {
crc = crc_table[(crc ^ buffer[i]) & 0xff] ^ (crc >> 8);
}
return crc ;
}
```

UTC_TIME type

Item	Field	Type	Length	Field Description
1	day	U8	1	Range: 1~31
2	month	U8	1	Range: 1~12
3	year	U8	1	Range: 0~255, starting from 2000
4	hour	U8	1	Range: 0~23
5	minute	U8	1	Range: 0~59
6	second	U8	1	Range: 0~59
Example	6 Byte, UNIT RTC time, GPS time both use this type, in UTC time standard. Transmission order is from small item to big. E.g.: 2014 Jan. 3rd 13:33:36, Decimal value: 03,01,14,13,33,36, Hexadecimal value: 0x03,0x01,0x0E,0x0D,0x21,0x24.			

GPS_INFO typeutc_time

Item	Field	Type	Length	Field Description
1	utc_time	UTC_TIME	6	Refer to utc_time
2	status	U8	1	Bit0~1: Location 00= Not fixed, 01/11= fixed.. Bit2: Latitude 1= N (north latitude), 0=S. Bit3: Longitude 1= E(east longitude), 0=W. Bit4~7: Reserved.
3	latitude	U32	4	Range: 0~90*3600000. Unit: MS Use with north and south mark (north latitude "+", south latitude " -")
4	longitude	U32	4	Range: 0~180×3600000. Unit: MS. Use with east and west mark (east latitude "+", west longitude " -")
5	speed	U16	2	Range: 0~65535. Unit: cm/sec(1 knot =1.852km/hour,1 knot =51.5cm/sec).
6	course	U16	2	Range: 0~3599. 1/10° Range: 0~359.9°
7	hight	S16	2	Range: -32768~+32767. Unit：0.1M Range: -3276.8~+3276.7M