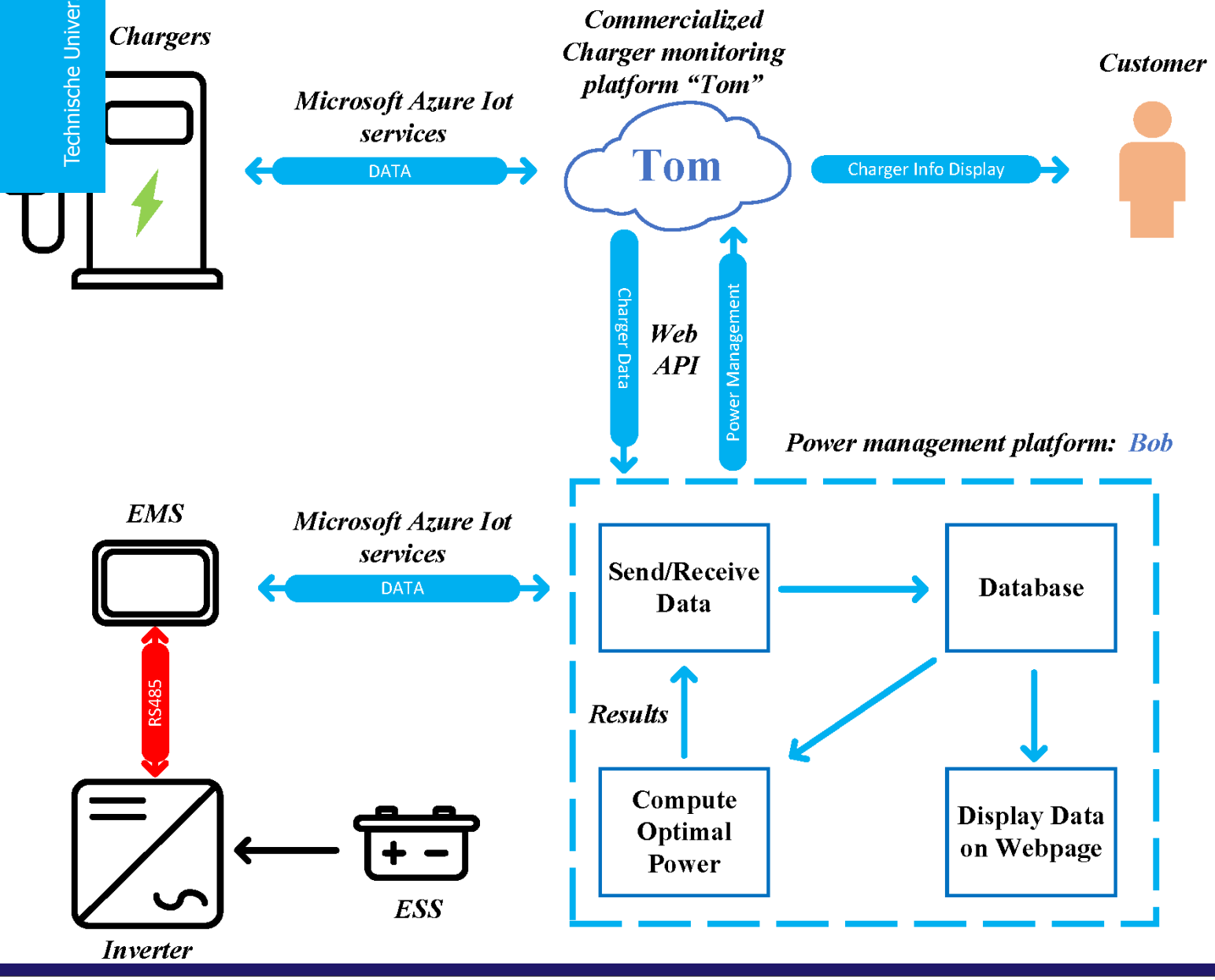
Cloud platform for EV charging management

Qixiang Xu

Technische Universiteit Delft



TU Delft Delft University of Technology

Challenge the future

Cloud platform for EV charging management

by

Qixiang Xu

in partial fulfillment of the requirements for the degree of

Master of Science
in Electrical Engineering

at the Delft University of Technology,
to be defended publicly on Tuesday October 27th, 2021 at 13:30 PM.

Supervisor:	Dr. ir. Zian Qin,	TU Delft
Thesis committee:	Prof. dr. ir. Pavol Bauer,	TU Delft
	Dr. ir. Zian Qin,	TU Delft
	Dr. ir. Pedro P. Vergara,	TU Delft

TU Delft Delft University of Technology

Abstract

Electric vehicle (EV) charging stations play an important role in the future development of the EV market. Uncoordinated charging will generate extra costs and bring unexpected stress to the grid. Multiple efficient charging strategies were proposed in past papers in order to solve the issues brought by uncoordinated charging. However, to put the algorithms into practical use, it is necessary to develop a platform for practical testing of the algorithms. In recent years, in the context of the gradual maturity of Internet technology, with the aid of embedded programming, the deployment of large-scale smart chargers on cloud-based platforms becomes possible. In cloud-based systems, the development, operational cost and system complexity are reduced compared with hardware PLC programming. Therefore, the cloud based platform is chosen as the testing platform for charging algorithms.

In the meanwhile, the energy storage systems combined with photovoltaic systems also can be used to lease the stress from the chargers. Hence, in this thesis, the ESS to cloud and its control strategy are implemented to reduce the impact from the chargers. In general, this thesis implements a cloud-based platform with the integration of the ESS, which is able to monitor the status of all the devices from the cloud, and distribute power to all devices with the aid of the charging algorithms on cloud.

To begin with, the different IoT solutions are discussed in introduction. Based on analysis and the experimental conditions, the most proper solution is chosen. Then, the details of cloud system structure is explained. Afterwards, the implementation procedure of the cloud platform is introduced by dividing the whole platform into different sections according to different functions on cloud. The contents include charger monitoring and control, charger grouping, ESS monitoring and control, database management.

Subsequently, two charging algorithms and an ESS control strategy are proposed. The theory of the algorithms and their function are introduced. Next, the interface design procedure on cloud platform is illustrated to show how the data is collected from the device to cloud, how the message is processed and computed on cloud. Based on above results, the simulation results are displayed to investigate the performance of the different control strategies on cloud in an ideal condition.

Finally, the whole cloud system and charging algorithms are validated and evaluated through the piratical experiment. The performance of the algorithms under the practical conditions are evaluated. The system cost and the delay are discussed. At the end of the thesis, the characteristics of the cloud based system are given based on previous analysis.

Acknowledgement

I would like to acknowledge and give my warmest thank to my supervisors Dr. Zian Qin, and Prof. dr. ir. Pavol Bauer for their insightful comments and suggestions.

Additionally, I would like to express my sincere gratitude to my daily supervior Dr. Zian Qin. His insightful comments and suggestions helps me to finding the way when I am lost during the project. With his help and supervising, I kept motivated for eight months since I have been working on this thesis.

I also would like to offer my special thanks to my colleagues from Third Place Energy B.V., with their help, my works can proceed smoothly.

I would like to extend my sincere thanks to my parents and my friends for their unwavering support and belief in me.

Qixiang Xu
Delft, September 2021

Contents

List of Figures

List of Tables

1

Introduction

1.1. Background

1.1.1. General background

Over the last few years, the sales of electric vehicles (EV) in European Union grows rapidly. From 2019 to 2020 sales of electric vehicles in Europe have almost doubled[1]. To fulfil the growing charging demands, and with the develop of fast charging technology, a large number of fast charging stations (FCS) have been built in recent years. In early 2021, the dutch government published a statistic of charging points in Netherlands, the statistical result is shown in 1.1[1]. According to statistics, from 2019 to 2021, the total charging points rose by 25%. Among these new charging points, the number of fast charging points increased by 825, the amount of growth is almost twice of 2019. However, if these newly established fast charging stations charge EVs without any control method, the large impact loads will be generated, which may brings additional issues to the power grid[2]; such as harmonic issue[3–5], unbalance voltage[6, 7], and peak loads [8].

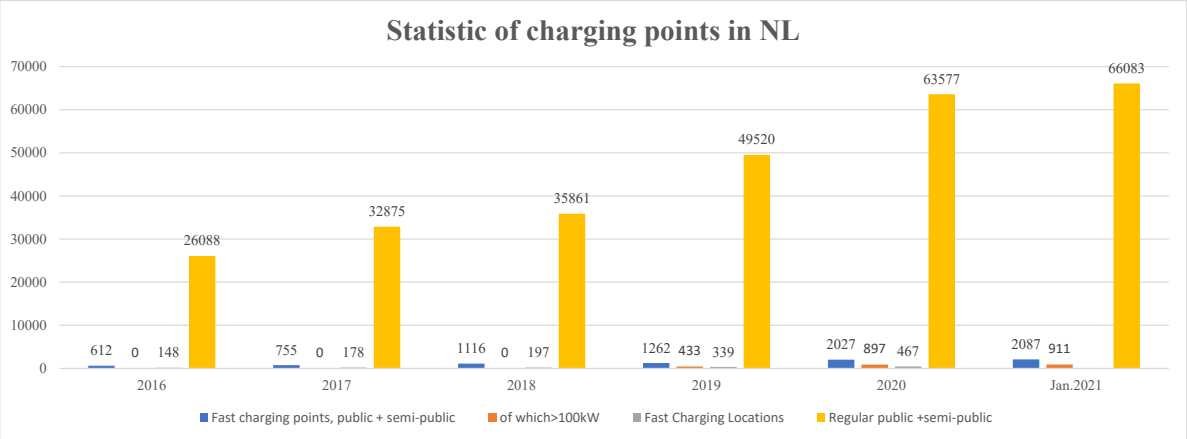


Figure 1.1: The Statistic of charging points in Netherlands from 2016 to 2021

1.1.2. Chargers with ESS

In the mean time, the energy storage technologies develop rapidly. Thereby, the cost of setting up a energy storage system is reduced. It is possible to integrate the ESS into the charging station. Furthermore, some government gives incentive to the charging stations for integrating more renewable energy to the station [9]. Besides, the charging station owner aim to gain more profit by running a charging station. Therefore, multiple optimization algorithms are proposed by past papers, such as [10, 11]. With the aid of these optimization algorithms, the profit of the charging station can be further increased. So, integrate the ESS into the charging station is the future trend and it can bring amount of benefits to the charging station owner.

1.2. Motivation and Objectives

1.2.1. Motivation

Currently, people are focusing on how to develop the more advanced and efficient algorithms for charging station, and most of the algorithms are validated through the simulation, not the practical test. There might be a lot of considerations has to be taken before putting an algorithm into piratical scenes. Hence this thesis will focus on the implementation of the whole charging behavior management system through the cloud solution.

Why manage charging behavior

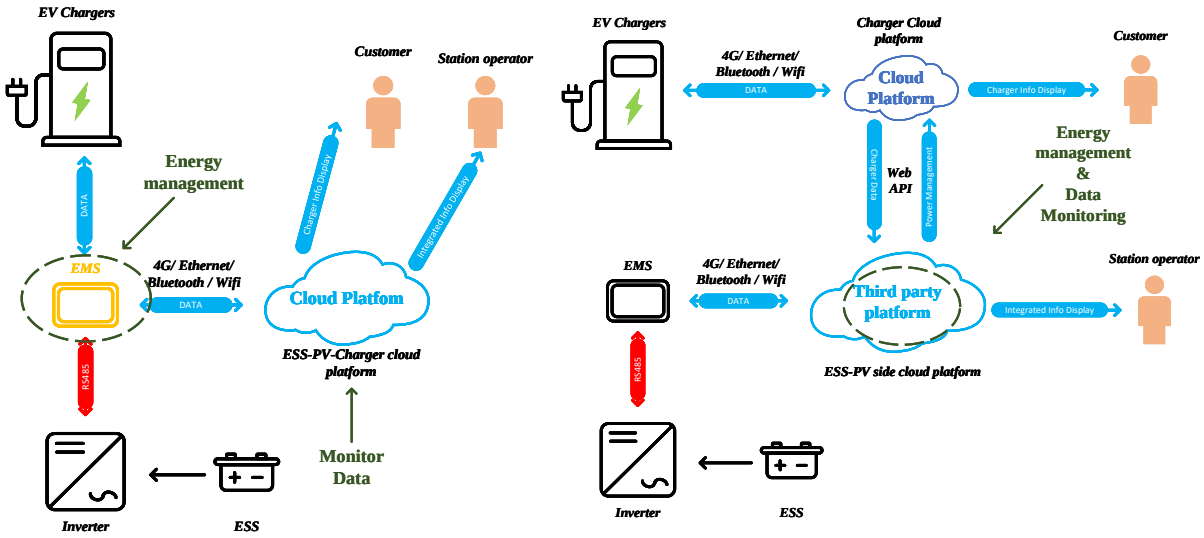
According to introduction in this section 1.1.2and 1.1.1, it can be noticed that the uncoordinated charging behavior and the uncontrolled ESS may bring multiple issues to the grid or to the charging station. Therefore, it is necessary to manage the charging behavior when EV is in charging in a charging station.

Why cloud-based

In the current market, the majority solution to the power management for chargers or ESS is through the EMS device, which will be placed at local site. This solution responses fast and reliable. However, the core of the EMS device is developed based on programmable logic controller, which may further increase the cost of the system. Furthermore, the local system is hard for maintenance and has limited scalability.

As for cloud solution, no extra component is required for the cloud solution so the overall cost can be reduced. Moreover, the cloud solution is easier to maintain, and the algorithms can be updated fast so that the development and maintenance cost will be reduced as well. Nevertheless, the cloud solution requires the network configuration, which may be unstable. But with the development of 5G or WiFi technology, this issue can be minimized. In addition, the cloud solution or the IoT technology is the future trends to the system, many companies are trying to embed the cloud platform into their systems, such as *Sungrow, Huawei*. Therefore, the cloud based platform is chosen to be the solution to this project.

There are two state-of-art cloud solutions to the charging management cloud platform, the one can be called *manufacturing execution system (MES) to cloud* and another one can be called *cloud interconnection method*, the structure of these two methods are shown in fig 1.2



(a) MES to cloud

(b) Cloud interconnection

Figure 1.2: Two cloud solutions

The MES to cloud method will put the charger and the ESS together to the EMS which is highlighted in fig 1.2a, in this method, the computation of the algorithms will be put at local EMS device, and the device will be connected to cloud directly. In this method, the cloud only provides the monitoring and

remote control function to the devices which are connected in this system. Since all the computation will be done in local device, the data traffic from device to cloud will be low. However,all the devices will be connected to the EMS device, this method requires better performance of the hardware and the system logic will be more complex which will greatly increase the system development cost and development period. This method is suitable for the companies which can provide complete charger-ESS integrated system to customer.

On the contrary, the second *cloud interconnection* method provides a completely different cloud structure compared with the MES method. This cloud interconnection method aims to dock the other charging platform developed by other companies. For further explanation, currently, there are many charger companies have developed their own charging platforms. However, if the customer wants to buy an ESS for the chargers, there will be no suitable platform for this situation. Hence, a third party platform is purposed in cloud interconnection method to solve this issue. This third party platform will use web APIs to collect the data from the charger cloud platform and send the commands to the charger platform through the APIs as well. Besides, the third party platform is able to monitor the ESS condition and manage the power of ESS. Furthermore, the optimization will be done on the third party cloud platform is based on the data collected from the ESS and the charger. Compare with the previous design, this design has lower system complexity, due to the fact that charger platform is provided by other company, therefore, the development cost will be reduced as well. Still, due to the existence of this charger platform, the system logic does not need to be changed which means the development period can be reduced significantly, and thereby, the versatility of this structure is better than the previous solution. Nevertheless, all the optimization calculation will be done through the cloud, the data traffic to cloud will be significantly increased. Hence, the data transmission rate has to be considered carefully.

In this thesis, the charging platform has been already provided by the other company. Hence the second method is selected to be the core structure of the whole system. The more detailed structure will be introduced in section 1.3.

1.2.2. Objectives

From a macroscopic point of view, this thesis is aiming to build a cloud-based EV charging management platform for fast charging stations. Therefore, a cloud-based platform will be created, the connection of all cloud-connected internet of things (IoT) devices will be implemented, and several economic charging algorithms will be proposed.

In general, the objectives of this thesis can be listed below:

1. Build a cloud-based platform with following functions.
    - The platform is able to collect and display the required real-time information. Such as inverter data, charger data, etc...
    - The platform is able to set the charging power of each chargers.
    - The platform is able to read and set the inverter parameters,and the charging and discharging power of ESS.
2. Design and implement charging algorithms to optimize the charging performance of the charging station.
3. Design and implement a peak shaving method to increase the power capacity of a charging station with the integration of ESS.

More details will be explained in section 1.3.

1.3. Project overview and approaches

The structure of this project can be found in 1.3. The structure is similar to 1.2b. The blocks with black color represent the devices which need to be monitored and controlled; The cloud-based technologies are pointed by blue blocks and arrows. In brief, the cloud-based platform will collect the data from the hardware devices, and the cloud will do the calculation to find the optimal power for each devices. Finally, the result will be send back to devices.

There are two cloud-based platforms: the first one is called "Tom", and the other one is called "*Bob*". "*Tom*" platform is a commercialized platform that can collect all charger data registered under this platform and can set the charging power of each charger, display the data on webpage, or pass the data to the other application through the web APIs. The "*Bob*" platform is the power management service to be built in this project which also can be called the third party platform with the reference to fig 1.2b. The functions in "*Bob*" are displayed in the dotted area of the block diagram. As shown in 1.3 dotted area, the "*Bob*" will receive the data from the "*Tom*" and the inverter. Then the useful data will be stored in the database of "*Bob*". After that, the "*Bob*" will use selected optimization algorithm to calculate the optimal power for the charger and ESS, at the meanwhile, the relevant information will be displayed on the webpage of "*Bob*". Finally, the results will be send back to the connected devices.

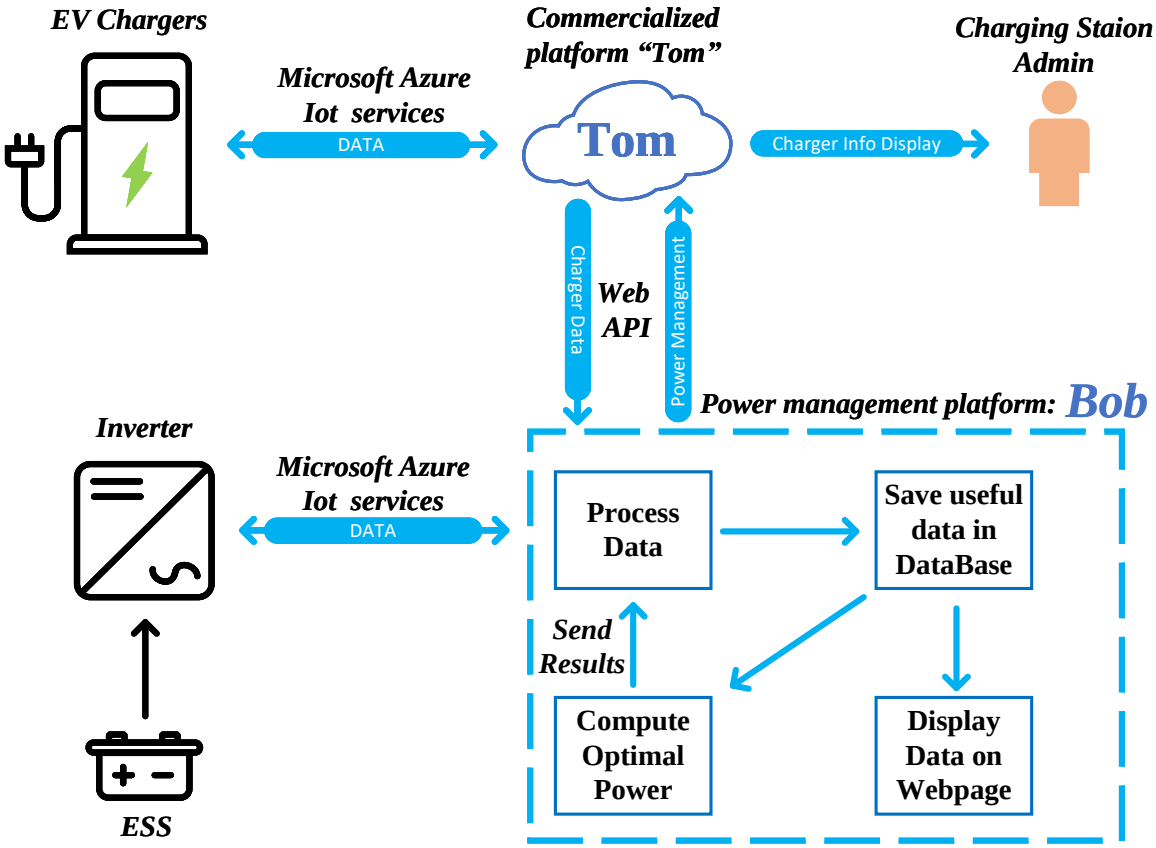


Figure 1.3: The structure overview of the project

In order to build the "*Bob*", the first step is to clarify the requirements. Next, sorting out all the external Web APIs and all the data requires power management algorithms because the appropriate usage of APIs and message format can improve the development efficiency and maintenance of the cloud platform. Afterwards, the basic functions of the "*Bob*" can be implemented. Such as the data processing from the other devices through the Web APIs or IOT Hub [12], and data sending to those devices.

After the basic functions are done, several coordinated charging algorithms are proposed and simulated on MATLAB. Then these algorithms will be implemented on built "*Bob*" and prepared for testing. To do the final validation, all hardware devices are connected and tested.

Finally, the experiment validation is done based on simulation results. The performance of each coordinated charging algorithms is evaluated by comparing the experiment results and the simulation results.

1.4. Project contribution

The thesis project has contribution in the implementation of a real cloud-based charging management platform and the real use of charging optimization algorithm. The contributions are listed below:

1. A platform for the EV charger and ESS power management is built. This platform is able to manage charging behavior along with the ESS in station; the platform also able to monitor the inverter and all data from chargers; The platform user is able to choose different charging algorithms from the cloud platform.
2. Two charging algorithms are implemented and tested through the experiment to prove the cloud solution is feasible. The fast charging by ratio and online coordinated charging algorithm and the performance of these algorithms are evaluated.
3. Different cloud solutions for different scenarios are suggested by evaluating the system cost and overall communication delay of the system.

1.5. Structure of the work

This thesis are divided into five chapters:

1. In chapter 1, the background, research objectives were introduced; the project overview and the approaches, the contributions were briefly described.
2. In following Chapter 2, the structure of the EV charging management cloud platform will be clearly introduced and analysed. The detailed information of the cloud platform such as Web API, UI interface and the hardware communication protocol will be mentioned as well.
3. In Chapter 3, the principle of each charging algorithms will be discussed. To show the performance of each algorithm, the simulation results and the comparison of these results will be presented in this chapter. The MATLAB CVX Tool is used to solve every optimization problems. Therefore, the detailed solving process of each optimization problems will be ignored in this chapter.
4. In Chapter 4, the test environment and the experiment result will be presented.
5. In Chapter 5, a conclusion will be made to summarizes the work and the achievements based on the contents of previous chapters. After the conclusion, the future works will be discussed.

2

Design & implementation of cloud platform

A platform that can manage each charger in FCS is crucial, as introduced in section1.3, the commercialized platform "*Tom*" supports several basic charger management functions. Such as EV charger data storage and display, remote charger charging power control and charger grouping. However, the power charging power of the charger can only be manually set by the administrator. Moreover, "*Tom*" platform does not support the management of inverter. Therefore, to build a complete platform demonstrated in 1.3, a power management platform "*Bob*" that can support different charging algorithms and the inverter management need to be built.

In this chapter, the basic introdurction of "TOM" platform will be made; the structure and the functions of "*Bob*" will be explained. The content includes:

- the introduction to commercialized platform "TOM" .
- the structure design and implementation of cloud-based power management platform "BOB"

2.1. Charger management platform *Tom*

2.1.1. The introduction of *Tom*

"*Tom*" platform is a commercialized charger management platform supported by the Third Place Energy B.V. [13]. Once the charger is registered on this platform and has an internet connection, the charger's data will be displayed on "*Tom*". Therefore system users are able to view the details of each charger registered on "*Tom*". The screenshot of one of the charger information details on "*Tom*" are shown in Fig.2.1. There are two titles in this screenshot: "*Charger Detail Dto Static Information*" and "*Real Time Data*". The former indicates the static information of the charger, such as the serial number of the charger, the software Version and the hardware detail of the charger, etc... The latter shows the dynamic data of the charger, for example, the charger status, charging power and vehicle details.

Furthermore, "*Tom*" also provides the grouping service. Hence each charging station can put their own chargers into a group, by doing so, the chargers can be managed by group. Fig 2.2 shows one of the group on registered on "*Tom*". the name of the charging station is "*test1*", which contains two chargers: "*testCharger4*" and "*testCharger2*", and the maximum charging power that this charging station can afford is $20kW$.

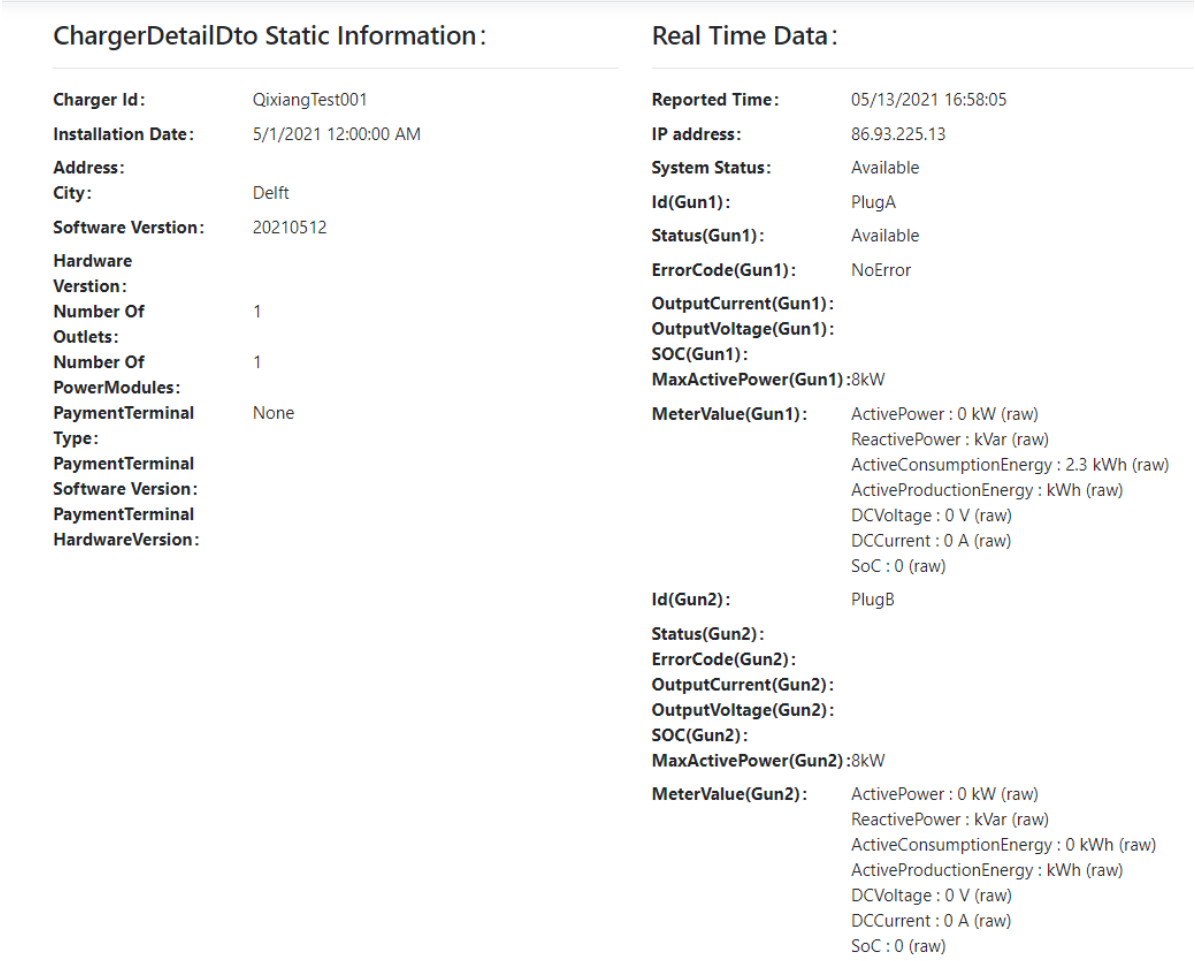


Figure 2.1: The details of one of chargers on "*Tom*"

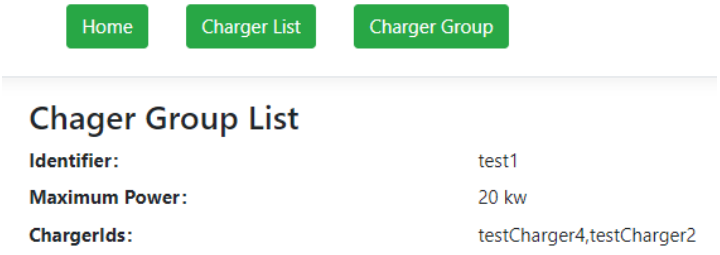


Figure 2.2: The grouping service on "*Tom*"

2.1.2. Tom APIs

As introduce in last subsection, "*Tom*" was not only able to display the data of the chargers on its website, "*Tom*" also provided some Web APIs for the other applications to call. Before developing proposed power management platform "*Bob*", it is necessary to conclude all Web APIs within a table for future use. Thus, the APIs from "*Tom*" and their functions are concluded in Table 2.1.

There are 6 interfaces provided by "*Tom*", the full url to access the each API is: *https://<address of Tom>/api/< request url>*. The first two interfaces use the GET method. By sending a GET request to "*Tom*", "*Tom*" will return a JSON list of all chargers' ID or return a JSON message with detailed information about a certain charger as response. Due to confidentiality agreements and space considerations, listing 1 only show part of JSON messages as example.

Table 2.1: The web APIs from Tom

All request url for interfaces start with **https://tpeportaldev.azurewebsites.net/api/**			
Request url	Function	Method	Body
TpeCharger/all	Get the ID of all chargers	GET	None
TpeCharger/details/<chargerId>	Get the details of the specified charger	GET	None
TpeCharger/setpower/<chargerId>	Set the maximum output power of the specified charger	POST	JSON
TpeChargerGroup/all	Get the ID of all charger groups	GET	NONE
TpeChargerGroup/Create	Create a new charger group	POST	JSON
TpeChargerGroup/Update	Edit existing charger information	PUT	JSON

```
Response from TpeCharger/all
  [
    {
        "chargerId": "QixiangTest002",
        "installDate": "05/01/2021"
    },
    {
        "chargerId": "QixiangTest001",
        "installDate": "05/01/2021"
    }
   ]
```

Listing 1: The JSON message example of Tom API Tpecharger/all

The example response of TpeCharger/all API includes a list, which is indicated by sign "[" "]", there are two items in this list, each item includes two messages: charger ID and install date. In fact, in real development, the developer is able to view all chargers ID registered on tom by deserilize this JSON message. Moreover, the developer can go further by calling the Tpecharger/details/<chargerId> API to view the details of the specified charger (<chargerID> needs to be changed to a specified charger ID). The response of Tpecharger/details/<chargerId> API includes the charger ID, status, location, maximum output power, current, voltage, vehicle SOC, software version, and hardware details.

The third API from Table 2.1 can be used to set the maximum output power of the specified charger by replacing the <chargerId> to specified charger Id. This POST method is used by this API. Therefore, a body must be included while sending a POST request to "*Tom*", the body should be in JSON format. listing 2 shows the POST body of setting the output power of charger "QixiangTest001".

```
Body of TpeCharger/SetPower/QixiangTest001
    {
        "ActivePower": [
        "Value": "30",
        "Unit": "kW"
        ],
        "ReactivePower": [
        "Value": "0",
        "Unit": "kVar"
        ]
    }
```

Listing 2: The JSON body example of Tom API TpeCharger/SetPower/<ChargerId>

```
Response from TpeChargerGroup/all
    [
     {
          "groupId": "test1",
          "chargerIds": [
              "testCharger4",
              "testCharger2"
          ],
          "maximumPower": {
              "value": "50",
              "unit": "kw"
          }
     }
     ]
```

Listing 3: The JSON body example of Tom API TpeCharger/SetPower/<ChargerId>

The last three APIs from Table 2.1 provide the view, as well as creating editing services to the charger group stored in "*Tom*". As mentioned in the last subsection, each charger group represents a charging station. In addition, the charging station can allocate its chargers into different groups for easier management. Hence the developer is able to call these three APIs for different purposes. The TpeChargerGroup/all uses GET method, similair to TpeCharger/all, it returns the list of all charger groups stored in *"Tom"*. The example is shown in list 3. The example list contains a charger group named "test1", also includes all chargers' Id within the group "test1", and the total maximum allowable active power of this group. 3.

As for the interface "TpeChargerGroup/Create" and "TpeChargerGroup/Update", although they use different methods, both methods require the body, the body format is in JSON format, consistent with the item format displayed in the listing 3. The interface "TpeChargerGroup/Create" will create a new charger group on *"Tom"*, and the "TpeChargerGroup/Update" will edit and update the existing charger group on *"Tom"*. Thus, with the aid of the *"Tom"* interfaces the information exchange between charger and proposed platform *"Bob"* can be achieved, which will be discussed in following sections.

2.2. The design and implementation of *Bob*

In this project, the Microsoft Azure Iot hub services is used to implement IOT functions, and the Microsoft official provides strong support for ASP.Net for their Azure Iot hub [14]. Therefore, the Model-View-Controller (MVC) framework [15] supported by *Asp.Net v2.2* is selected as the frame of the "*Bob*".Hence, before starting the detailed design procedure, the basic functioning theory of the MVC framework needs to be introduced. Fig 2.3 shows a brief structure of the Asp.net MVC framework.

The essence of cloud platform is web application, and the web application can be created based on Model-View-Controller framework [16]. It splits a web application into three components: model, view and controller.

- Model: The Model component stores all data format and manage the data transferring between the View and the Controller. Furthermore, the algorithms and database management are included in Model component as well.
- View: Base on the data provided by Model component, View component displays a user interface on web page.
- Controller: The Controller component processes the request sent from View component and gives the response to View or Model component. Therefore, the controller can be treated as the brain of the web application.

As for "*Bob*", the View component will handle the UI of the platform; the the database management,

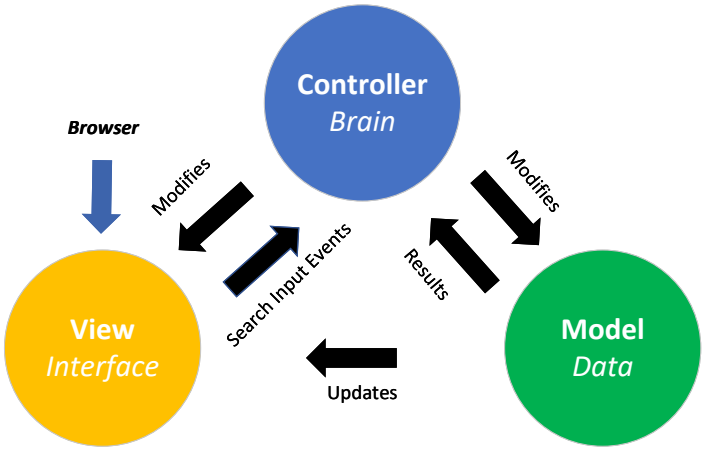


Figure 2.3: The ideal structure of "*Bob*"

algorithm implementation, and the definition of all data format will be finished in Model component; The controller will implement all the external and internal interfaces on "*Bob*".

2.2.1. The structure overview of *Bob*

As mentioned in section1.3,at the beginning of designing the platform "*Bob*", it is necessary to sort out the parameters and the web APIs required by the "*Bob*".To clarify the parameters and interfaces required by "*Bob*", the first step is to define the functions that need to be implemented on "*Bob*". According to objectives introduced in section 1.2.2, the functions can be specified in fig 2.4.

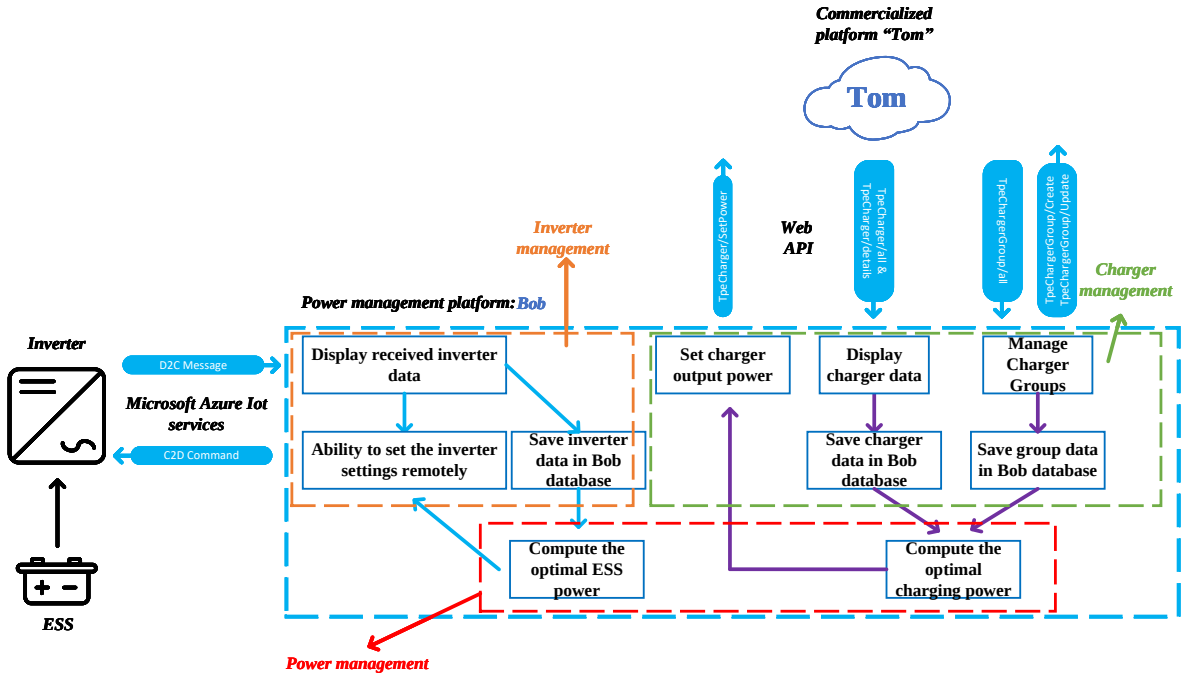


Figure 2.4: The ideal structure of "*Bob*"

According to figure showed in fig 2.4, the design procedure of "*Bob*" can be divided into three parts by different functions: The first part is called charger management which is encircled by green dotted line; The second part is the inverter management part, encircled by orange dotted line; The last part is power management, the implementation will be introduced in next chapter, because the optimization algorithms will be discussed in that chapter.

2.2.2. Charger management

From the fig 2.4, it can be seen that the different APIs from "*Tom*" support different functions on "*Bob*". Therefore, the function blocks can be implemented separately.

Display charger data on *Bob* web page

To display the charger data, the interface *TpeCharger/all* and *TpeCharger/details/<chargerId>* will be used. The MVC structure design and the relationship between each blocks can be summarized in an UML class diagram, fig 2.5.

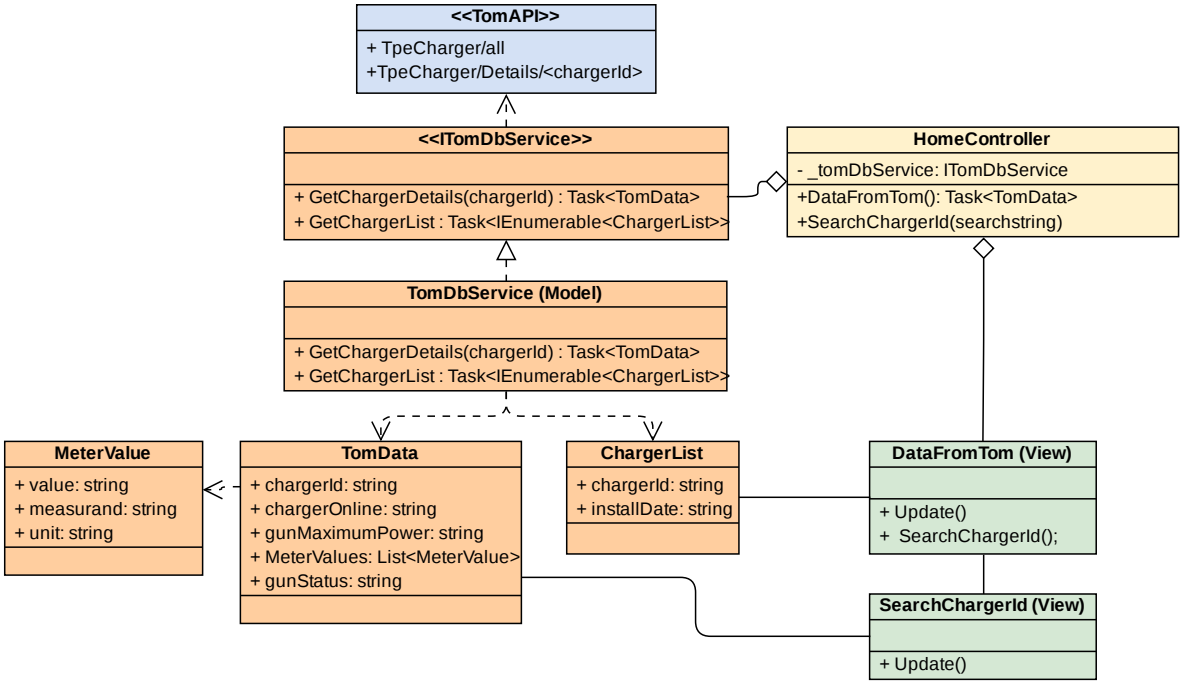


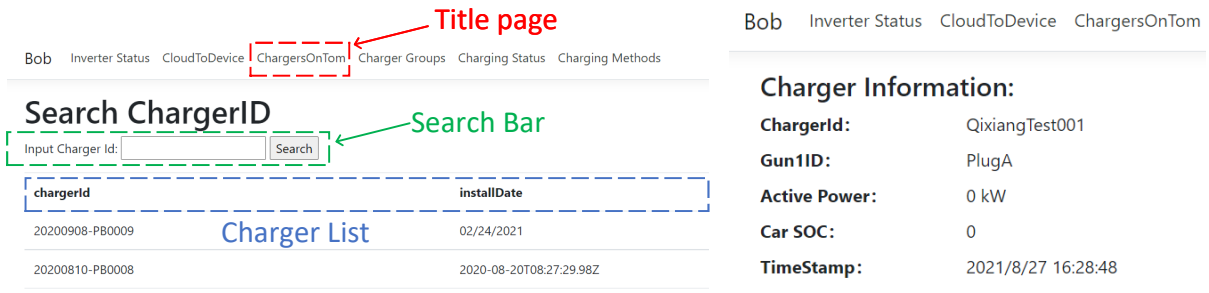
Figure 2.5: The UML class diagram of function: "Display charger data"

In order to better distinguish the subordinates of each block, the blocks are colored with different colors in fig 2.5. Each block represents a class or an interface, the interface is indicated by the sign: "<< >>". There are four components in this diagram: API from "*Tom*", Model, View, Controller. The block marked as blue represents the APIs from "*Tom*", the first layer defines the type of the block. The second layer of the interface block defines the methods (or operations) of the interface, the plus sign in front of the method indicates the defined method is public. So, as displayed in fig 2.4 and the content in section 2.1.1, these two methods are the APIs provided by "*Tom*". To process the data transmitted in these APIs, an interface is defined in Model component called "*ITomDbService*", and all blocks which belongs to the Model component are marked as orange. The arrow between "*TomAPI*" and "*ITomDbService*" means the "*ITomDbService*" is the dependency of the newly defined interface "*ITomDbService*". In this interface, two methods are defined to deserialize the JSON message from the response of the "*Tom*" API: One is to get the detailed information of a specified charger called "*GetChargerDetials()*", and the other is to obtain all charger IDs on "*Tom*" called "*GetChargerList()*". After the defining interfaces, the function of these interfaces needs to be implemented. Hence, another class "*TomDbService*" is defined to implement the interface "*ITomDbService*". Since the "*ITomDbService*" defined two methods, two methods with the same name will be defined in class "*TomDbService*". These methods are implemented by the classes which are connected below the "*TomDbService*" class. For further explanation, the class "*TomData*" and "*ChargerList*" define the data format that will be used by the methods defined in class "*TomDbService*". Once, the methods defined in class "*TomDbService*" can be successfully implemented, the interface "*ITomDbService*" will be fully functional.

Due to the characteristic of the MVC framework, the interface need to be scheduled correctly by the controller. Thus, a controller component "*HomeController*" is defined and colored by yellow, and this controller is partially aggregated by the interface "*ITomDbService*". Class "*HomeController*" has

three layers: the first layer indicates the class name; The second layer instantiate an object of "*ITomDbService*" interface, the methods defined in "*ITomDbService*" interface can be called through this instance object and the minus sign in front of the " *_tomDbService*" means the attribute of this object is private. The blocks in green represent the View components, and similar to the relationship between the interface and controller, the *HomeController* is partially aggregated by the View *DataFromTom*. The view *DataFromTom* has two methods, the *Update()* method will display the charger list recived from *TomAPI* with the format of class *chargerList*. The *SearchChargerId()* method provides a search bar for the users to search and views the specific information of the charger. Another View *SearchChargerId* defines a method *Update()* to display search results.

With the aid of the UML class diagram, the function: "Display charger data on "*Bob*" web page" can be implemented, the results are shown in fig 2.6. By clicking the label *chargersOnTom* on the "*Bob*" website, the chargers on "*Tom*" will be displayed on this page. In addition, the search bar can be used to view the specific charger data, the example is shown in fig 2.6b.



(a) The View of charger List

(b) The View of charger detail

Figure 2.6: The view of chargers on "*Bob*"

Charger group management on *Bob*

Move to the implementation of function*charger group management*. According to fig 1.3, from the "*Tom*" side, there are three APIs will be used to fulfil the function of charger group management. Similar to last section, an UML class diagram will be used to explain how this function is implemented.

The structure of the charger group management function is clearly shown in fig 2.7. Same with the previous class diagram fig 2.5, the MVC components are marked by three different colors: Orange, yellow and green. The blue block shows the APIs selected from "*Tom*" to implement the charger group management. There are three APIs: View all, Create and Update. Accordingly, the charger groups should be displayed, and able to create or edit on "*Bob*". By referring the design procedure introduced in the first part of 2.2.2, an interface should be defined to process the data transmitted by "*Tom*". At the meanwhile, the charger group management function is aggregated to charger management, which means all the functions under the charger management will share a common interface. Thus, the new operations will be defined in interface *ITomDbService* which was defined in last part. There are three newly defined functions: *GetChargerGroupList()*; *UpdateChargerGroup(chargerGroup)*; *CreateChargerGroup(chargerGroup)*. The first method will be used to deserilaize the JSON message from the response of the *TpeChargerGroup/all* to get the list of charger groups stored in database of "*Tom*". The other two methods will be used to edit and create the elements stored in database of "*Tom*" according to the *PUT* and *POST* functions provided by *HTML*. Subsequently, the implementations of these interfaces are defined in class*TomDbService*. The message format of the charger group is defined in class *ChargerGroup* as the dependency of the *TomDbService*. There are three attributes in the *ChargerGroup*: groupId shows the name of the charger group; chargerIds contains all chargers in this group; The maixmumPower is a class which includes two arrtibutes, the value and the unit, this class will record the maximum power that the charging group can output.

In order to distinguish the scheduler of different functions, a new controller component *ChargerGroupController* is defined. This controller has one private attribute and five methods. The private attribute " *_tomDbService*" is an instance object of the interface *ITomDbService*. Its role is the same as that of the object in the *HomeController*, and they are all instantiated objects in order to be able

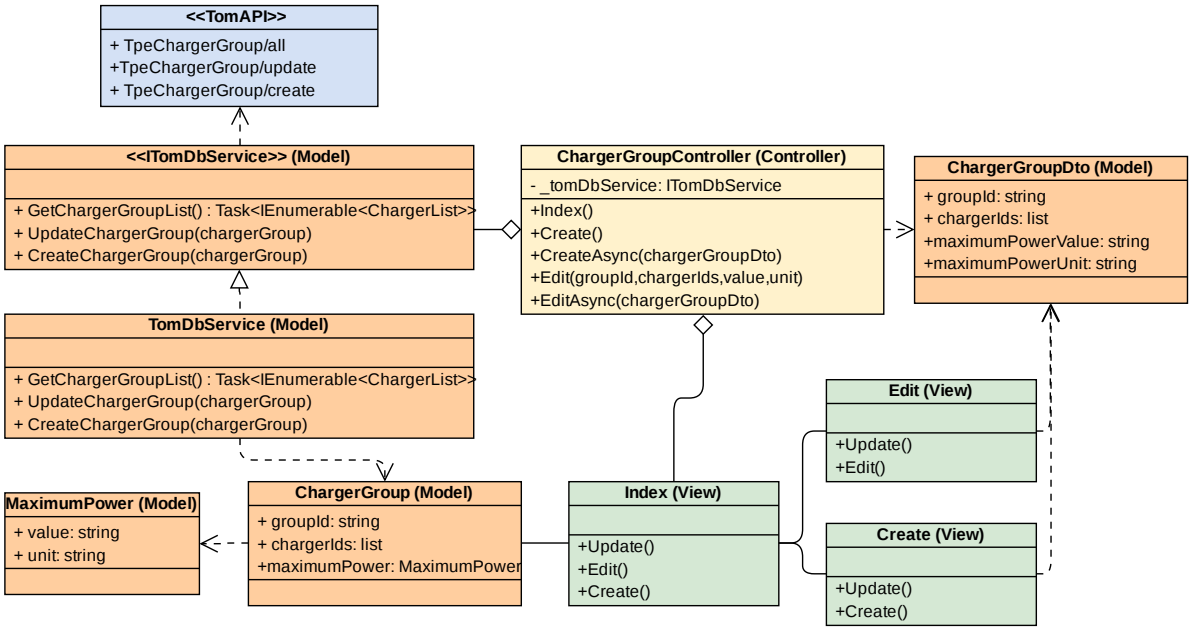


Figure 2.7: The UML class diagram of function: "charger group management"

to call the functions in the interface. To clarify the connection between controller and views, a list describes the meaning and the content of these five methods and their relationship between the View components is as follows:

- Index(): This operation triggers the method *GetChargerGroupList()* defined in interface and get the return value (Charger group list) of that method. Then parse the return value and transfer the result to the View component *Index*. The *Update()* function in View *Index()* will makes the received data to be visualized and update it to the web page.

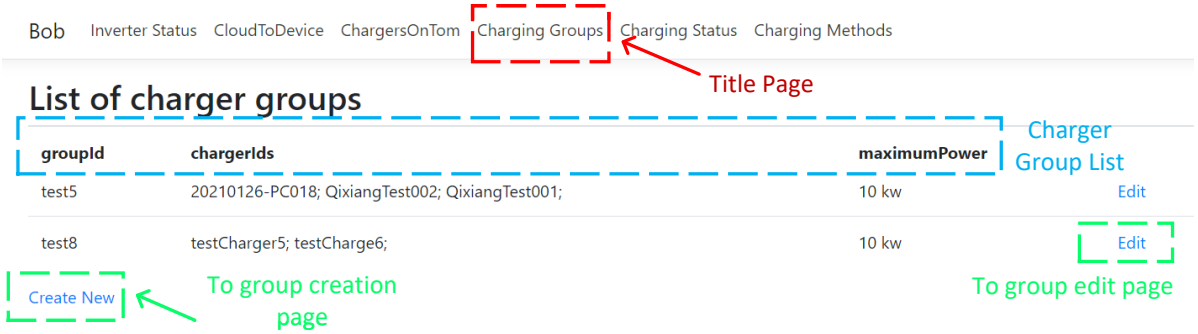
- Create(): This operation will be used as a trigger. In more detail, there is an operation *Create* defined in View *Index()*, this creates operation will create a "create new" button on the *Index* page. Once this button is clicked, the *Create()* method defined in controller will be triggered to finish the page jump action, the web page will be redirect to the View *Create*, which supports for the creation of the charger group.

- CreateAsync(chargerGroupDto): This operation has an argument called *chargerGroupDto*, the *chargerGroupDto* is a class defined to make the page visualization more convenient. This operation will be used as a trigger as well: Once the pager is redirected to group creation, and the user finished the input and click the create button, the *CreateAsync(chargerGroupDto)* will be triggered to POST the input to "*Tom*" API.

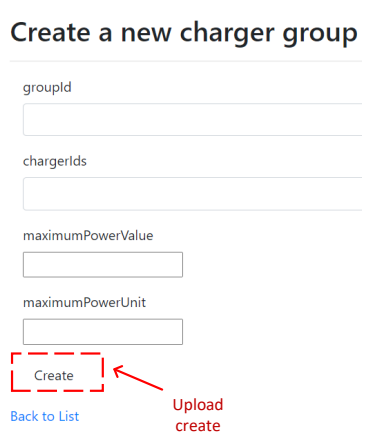
- Edit(groupId, chargerIds, value,unit): This operation will be used as a trigger, the operation theory is same as the *Create()* method.

- EditAsync(chargerGroupDto): This operation is similar to *CreatAsync(chargerGroupDto)*, this operation will be triggered once the user want to upload the changes by clicking the edit button on Edit View. The changed data will be PUT to "*Tom*".

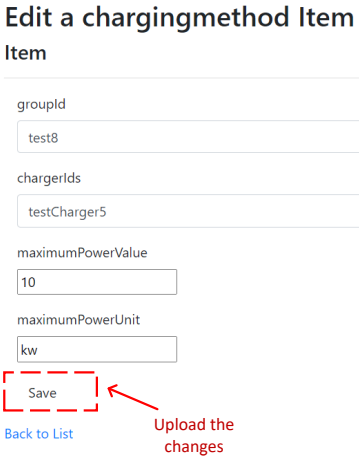
After implementing all the methods introduced above, the group management function is visualized on "*Bob*". The results are shown in fig 2.8. By clicking the label *Charger Groups* on the "*Bob*" website, the charger groups on "*Tom*" will be listed on this page. The create and edit button can be used to create or edit the charger group, the example is shown in fig 2.8b, 2.8c. Moreover, the create and edit functions has been validated as well.



(a) The View of charger group List



(b) The View of charger group creation



(c) The View of charger group edit

Figure 2.8: The view of charger group management on "*Bob*"

Send power change request from *Bob*

According to fig 2.4, the "*Bob*" should be able to send the power change request. The value of the power will be determined by the *Compute the optimal charging power* function on "*Bob*". Furthermore, "*Tom*" leaves an API to process the power change request. Therefore, this sends power change request function only needs to process the power sent by other functions and POSTs the data to "*Tom*". Fig 2.9 shows the UML class diagram of this function.

In order to POST the power change request to "*Tom*", A method called *PostPowerToTom(powerConfig)* is defined in interface *ITomDbService*. Subsequently, this method is implmented in class *TomDbService*, and the format of the message to be sent to "*Tom*" is defined by class *PowerConfig*. As motioned in the above paragraph, to put this method in use, the *SetPowerTrigger(changedCharger)* is defined in *PowerLimitController*. Consequently, once the *SetPowerTrigger(changedCharger)* function is called, the *PostPowerToTom(powerConfig)* will be executed to POST the calculated charging power to *"Tom"*. In addition, there is no View component in this diagram, because this controller is defined as an API controller, the API name is: *api/powerConfig*. This controller is created to support the charging algorithms on "*Bob*". Hence, the more details about this API controller and its methods will be explained in next Chapter. In addition, there should be a database to store the charger information and charger group data. So that the charging algorithm can use these data to perform calculations. The establish of database on "*Bob*" will be illustrated in subsection 2.2.4.

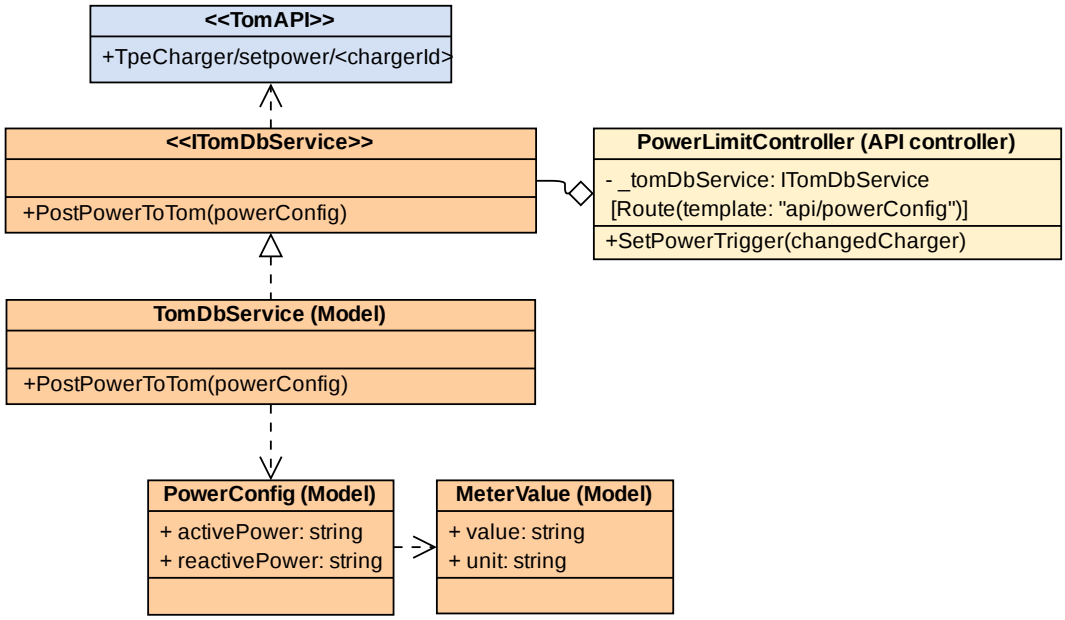


Figure 2.9: The UML class diagram of function: "send power change request"

2.2.3. Cloud-based inverter management

Based on the objectives of this thesis, the "*Bob*" should be able to view and manage the inverter parameters remotely. In this thesis, the *Micorsoft Azure Iot*[12] is used as the bridge between the device and cloud platform. The inverter used in this project is the R10KH3 $10kW$ PV-ESS hybrid inverter from the ShenZhen Megarevo Technology Co., Ltd. The detailed inverter technical parametrs will be provided in Chapter 4, in this section, only the IoT procedures will be introduced. Fig 2.10 shows the relationship between "*Bob*" and the inverter.The inverter has two COM port for data transmission, but has no ports for network cable. Since the inverter has no internet connection, an extra component is required to obtain the internet connection. Furthermore, the original screen on the inverter is too small to use, to improve the user experience, it is necessary to add an extra large screen for users to operate. Therefore, a 12 inch ARM based industrial computer from Chipsee is selected, the more details about this screen can be found in chapter 4. This ARM based computer provides the WIFI or 4G connection to the network and the inverter data can be read/write through the COM port on the inverter side. Hence, the inverter to cloud can be implemented by appropriate coding on industrial computer and the cloud platform.

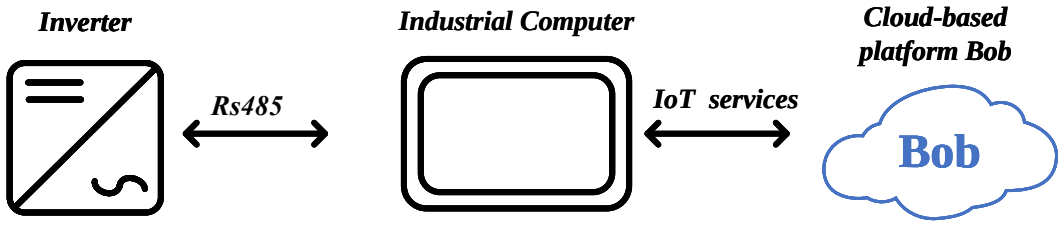


Figure 2.10: The block diagram of the inverter to cloud

A more detailed block diagram that illustrates the IoT procedures of the software programs can be found in fig 2.11. According to the block diagram, there are two software applications, which will be implemented in the industrial computer: One is the QT based application, this application will collect/send the data/instructions from/to the inveter, then display the required data on the screen of the industrial computer and send the data to JAVA program through the named pipe; The other one is the Java based application, this application is used to establish the connection between the industrial computer and the Azure Iot Hub (cloud side). Further, after the connection is established, this

application will read the data from the named pipe and send it to IoT hub. Vice versa, this application is also able to write the data into the named pipe once the cloud to device command has been sent. Next, there is a middleware between the industrial computer and platform "*Bob*". It is marked by blue dotted line with the name *Azure Iot services*. In this middleware, it has two parts: *Azure events* and *Azure functions*. The role of the *Azure events* is the trigger. This trigger holds the connection between the device and the cloud, and will be triggered once there is an message to be processed to device or cloud. Furthermore, the *Azure functions* is associated to *Azure events*, the *Azure events* will send the raw data received from the device to *Azure functions*, the functions defined in *Azure functions* will parse the raw data sent from *Azure events* and then store the parsed data into the database of "*Bob*". Finally, the rest functions on "*Bob*" will extract the data stored in data base and display it on the webpage of "*Bob*". The detailed implementations are shown in following paragraphs. This section is only focusing

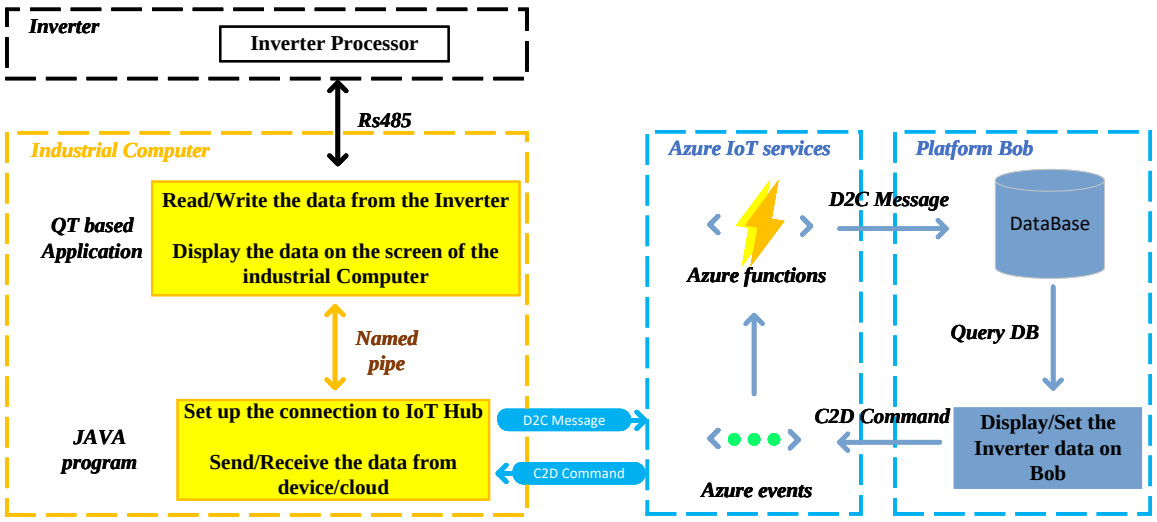


Figure 2.11: The software block diagram of the inverter to cloud

on the implementing the IoT function. Therefore, the implementation of QT based application and the analysing of the protocol between the inverter and the industrial computer will not be discussed in this section.

Java program

As demonstrated above, this Java program will be embedded to industrial computer to act as the medium between the device and the cloud. To further illustrate the structure of this program, the UML class diagram is shown in fig 2.12.

Due to the characteristic of the JAVA, there will be one public class. In this program, *SimulatedDevice* is the public class defined in this class, it has four private attributes. These four attributes define the characteristics of this java program, the details can be found in table 2.2. In addition, there are some private classes that are defined inside the class*SimulatedDevice*. Thus these private classes are associated to *SimulatedDevice*. The use of these private classes is explained in list below.

Table 2.2: The variables in class SimulatedDevice

Variable	Function
connString	The connection string to IoT Hub
protocol	The communication protocol between the cloud and device
testInv1	The Object name of the device
telemetryInterval	Time interval for sending telemetry data

- 1. *InverterData*: This class is defined to process the data received/sent from/to the inverter. Therefore, the declared variables represent the inverter parameters. the function *getPipeData()*

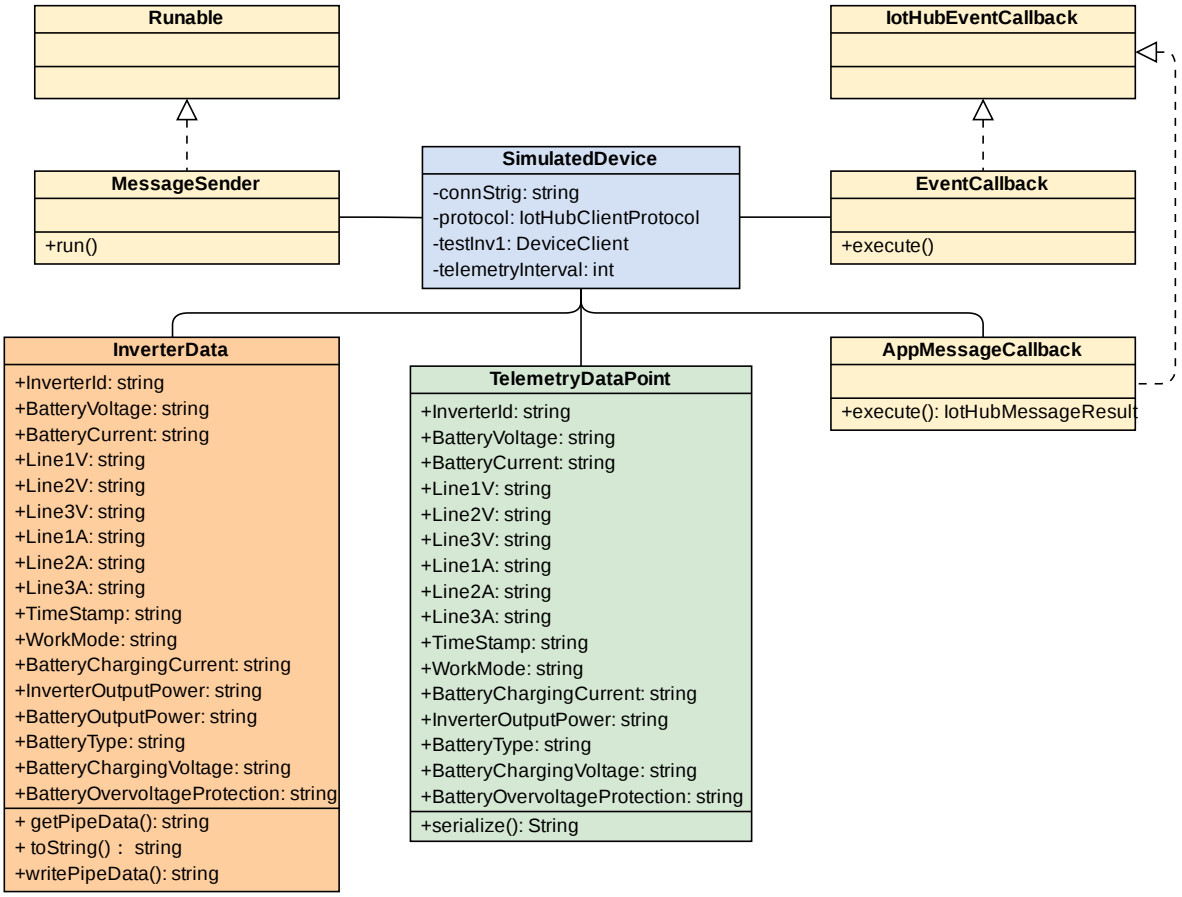


Figure 2.12: The UML class diagram of Java program

is used to retrieve the inverter data written in the pipeline by the QT program. Similarly, *writePipeData* is used to write the cloud command into the pipeline. The *toString()* is an overridden function of function *toString()*. This newly defined *toString()* function will convert a JSON object into a string.

- 2. *TelemetryDataPoint*: This class defines the final format of the message to be sent to the cloud. Hence, the declared variables represent the inverter parameters, which are the same as the variables defined in *InverterData*. The operation *serialize()* will convert a string to a JSON object.

- 3. *AppMessageCallBack*: This class is defined to process the cloud to device commands, this class implement the father class *IoTHubEventCallback*. The further details of *IotHubEventCallback* can be found in [17]. The method *execute()* will be triggered if a cloud to device command is sent. Then this method will process the recived command and write it into the pipeline. Finally, it returns the communciation result. If the message is successfully received and written into the pipeline, the "OK" will be returned.

- 4. *EventCallback*: This class is the implementation of the father class *IotHubEventCallback*. Similar to 3, an operation defined in this class is called *execute*. This operation will print the acknowledgement received from IoT Hub for the telemetry message sent. If the message is sent to cloud, the "Message sent successful" will be printed on console.

- 5. *MessageSender*: This class is the implements of the father class Runable[18]. *Run()* is defined as a function in this class, *run()* will use the methods defined in class *InverterData* and *TelemetryDataPoint* to retrieve the data from the pipeline and then send the processed data to cloud by a certain time interval. The time interval is defined in public class *SimnulatedDevice*

Azure IoT Services

From the fig 2.10, it can be seen that the messages need to be processed by the middlewares provided by Azure IoT Hub. Fig 2.13 shows the screenshot from the Azure IoT Hub. The defined Azure event is *IoTHubMessages*, this trigger connects to a azure function. Azure IoT hub provides an online editor to edit the function on Azure IoT hub. The detailed code of this function can be found in Appendix B. In this function, the recived JSON message from the device will be deserilized and stored to cloud database. There are two databases connected to this function. One is called *testInvDB* and the other one called *outputBlob*. The structure of *testInvDB* is CosmosDB, this database is used for real time data querying and storage. The other one is based on the Azure blob storage. This storage will be used to store the history data of the inverter.

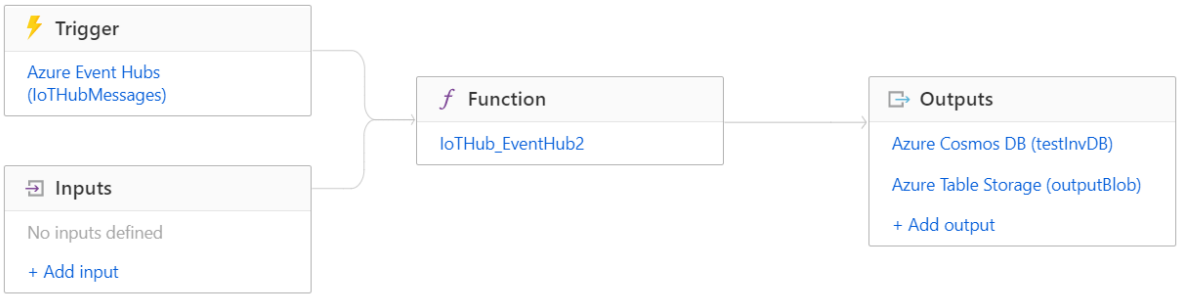


Figure 2.13: The integration of IoT services

Inverter data remote management on *Bob*

The remote management includes the display the inverter data on "*Bob*" and change the inverter settings from "*Bob*". As displayed in fig 2.11. To display the inverter data on "*Bob*", the real time data need to be read from the database. Moreover, the Azure event should be able to receive the command sent from cloud. To further illustrate how remote management is implemented on "*Bob*", an UML class diagram will be displayed below in fig 2.14:

As mentioned above, the inverter data will be stored in database of "*Bob*". To display the data on the website, the inverter data needs to extracted from the database with the interfaces defined on "*Bob*". Therefore, an interface *ITestInvDbService* is defined. Two operations are declared in this interface, one is *GetItemAsync()* and the other one is *GetStaticItemsAsync()*. Their implements are defined in class *ITestInvDbService*. This class has one private variable *_container*, this variable will be used to extract the data from the specific database. Moreover, this class implmenets three functions:

- 1. *TestInvDbService(dbClient,databaseName,containerName)*: This function will returns a reference to a container object, the database cannot be accessed without this function.
- 2. *GetItemAsync(id)*: This function is the implementation of the interface *GetItemAsync()*. It has one argument *id*, by pass on this argument into the function, the function will return an object from the database with the same id. The object extracted from the database can be deserilized with the format showed in class *InverterData*.
- 3. *GetStaicItemAsync*: This function will return all the objects saved in database as a list for user to iterate. However, this project only has one inverter. Thus, this function is designed for future use.

Similar to previous sections, a controller is needed to call the methods in the interface. There are four methods will be used in this controller. The explianation of each function can be found in list below:

- 1. *Index()*: Within this function, *GetItemAsync()* is called. Furthermore, this function connects to the View of *Index*. *Home* means the homepage of the website. Therefore, once the website is opened, this *Index()* function will be called and then *GetItemAsync()* will extract the inverter data by finding the specified inverter Id.

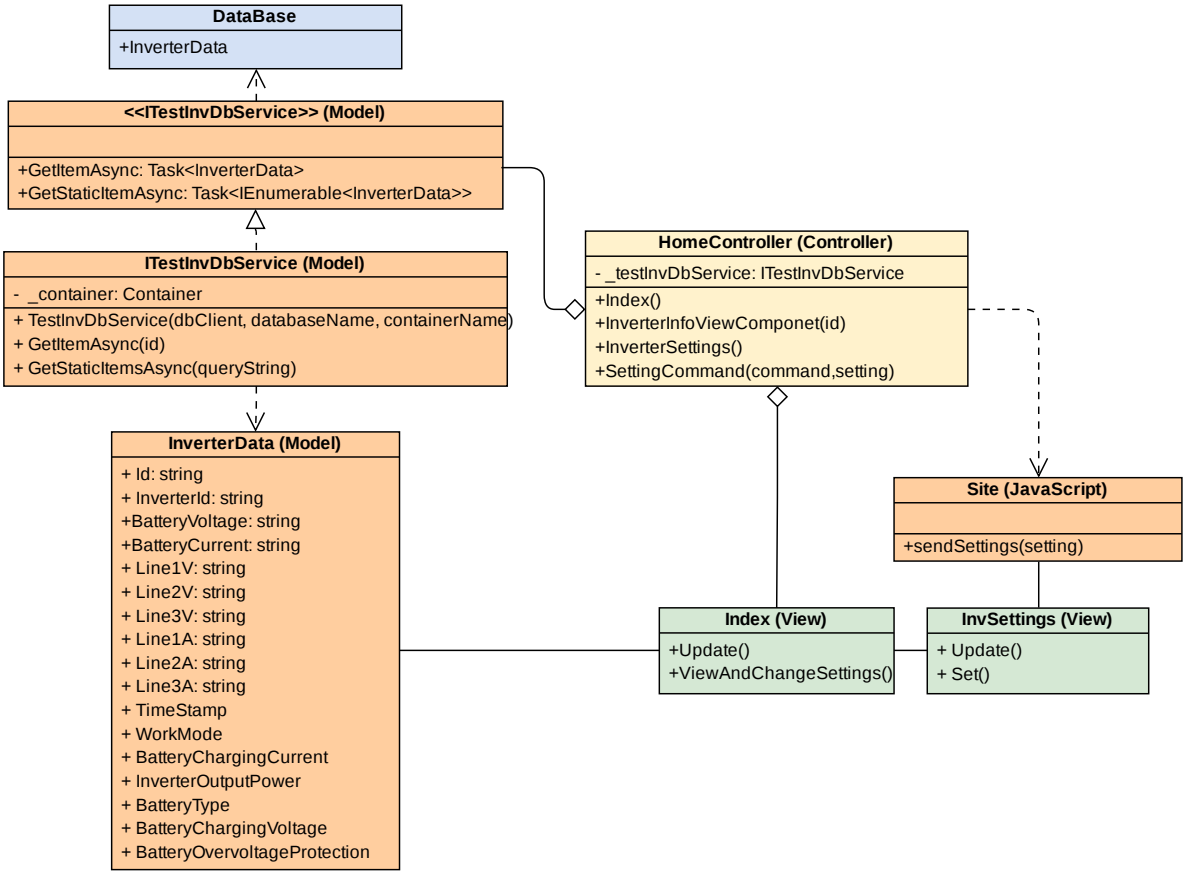
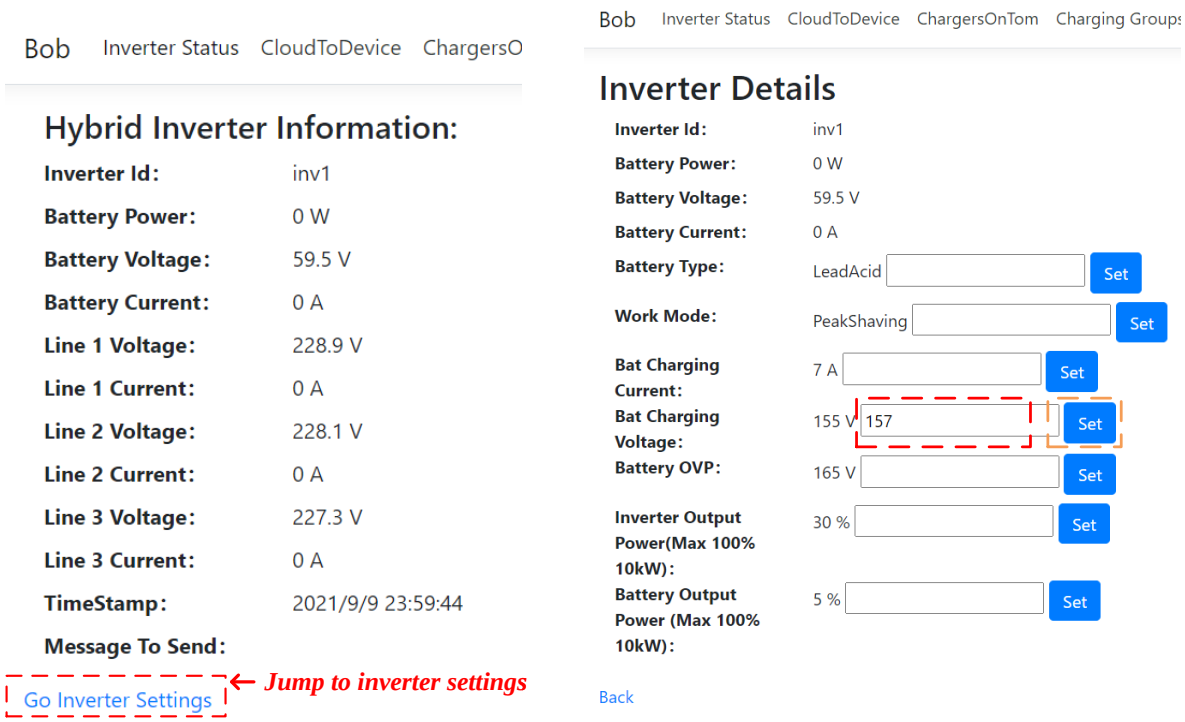


Figure 2.14: The UML diagram of Inverter management

- 2. *InverterInfoViewComponent(id)*: To display the data in real time, the data displayed on View has to be refreshed periodically. Hence, this function uses the script from the JavaScript. By calling this function in View *Index*, the webpage can be refreshed periodically.
- 3. *InverterSetting*: This function implements the push button defined in View *Index ViewAndChangeSettings()*, by clicking the "Go inverter settings" button, this function *InverterSettings* will be called and the page will jump to View *InvSettings*.
- 4. *SettingCommand(command,setting)*: This function sends the received settings from the website to the device. For further explanation, the View *InvSettings* defines multiple push buttons for different inverter options. These push buttons are implemented by the *Set* function defined in View *InvSettings*. To distinguish which button is triggered, an extra method is defined in static file *Site*, this method is written by JavaScript. This *sendingSettings* method will add an extra attribute to each push buttons, called *setting*. For example, if the user wants to set the charging voltage of the inverter to 156V, once the "set" button is clicked, the *sendSettings(setting)* will be called automatically and add a tag "BatChargingVoltage" as setting along with the value 156V as the command. Afterwards, this JavaScript function will post the setting and the command to the *HomeController* to trigger the function *SettingCommand(command,setting)*. Once this *SettingCommand(command,setting)* function is triggered, this function will send the recived arguments command and setting to the device through the Azure IoT service.

By implmenting the UML class diagram above, the inverter management can be implemented, the screen shot of the website can be seen in fig 2.15. The data will be updated every seconds. By clicking the button *Go inverter settings*, the user is able to view change the settings of the inverter. The user can input the newer value in to the text box as shown in circled red line in fig 2.20c, then by clicking the "Set" button, the command will be sent to device.



(a) The View of Index

(b) The View of inverter settings

Figure 2.15: The view of Inverter management on "*Bob*"

2.2.4. Database of *Bob*

In order to record the data from the inverter and the data from the charger for calculating, it is necessary to establish the database on "*Bob*". This section will introduce how to establish the database by using tools provided by Azure and what types of data will be stored in database.

Azure CosmosDb

The Azure CosmosDb is selected to be the database on "*Bob*". CosmosDb is commonly selected as the solution to the IoT. This is because the CosmosDb is able to get the telemetry data from the device at high rates and its queries has low latency and high availability [19]. The CosmosDb can be created through the Azure portal Fig 2.16 shows a typical structure of the CosmosDb.

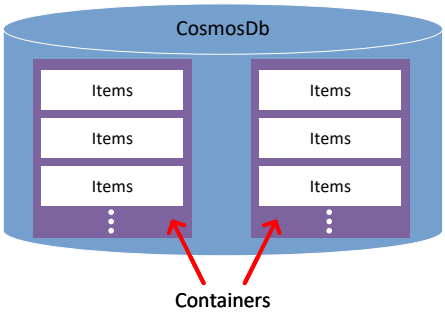


Figure 2.16: The structure of CosmosDb

The CosmosDb may have multiple containers, all the data will be stored in the container in the form of item. For example, in this thesis, a CosmosDb is established with the name of *evChargingDatabase*, as shown in fig 2.17, this CosmosDb has two containers: *dynamicData* and *groupChargingMethodData*. The past charger data will be stored in container *dynamicData*, each item represents a charger.

The charging methods will be stored in container *groupChargingMethodData*, each item represents a charger group. The data format stored in database are shown in list 5 and 4. The data stored in these two containers will be used for future optimization algorithms.

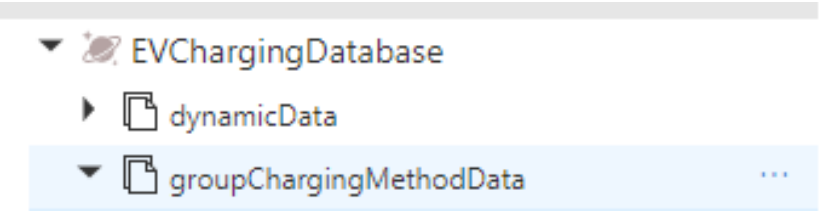


Figure 2.17: The example of CosmosDb

Furthermore, as mentioned in last section, the inverter data will be stored in a database as well. To reduce the cost of using the IoT service, the inverter data is stored in another existed CosmosDb, this CosmosDb belongs to "*Tom*", called *ChargerTestData*, a new container is created called testInvData, the format of inverter data is similar to the list 4.

```
{
    "id": "fda36c71-1484-494d-a271-3959ca4f4416",
    "groupNum": "test5",
    "groupChargingMethod": "OLCC",
    "CompensatingPower": "10kW",
    "ESSMethod": "PeakShaving",
    "_rid": "XYxyAONhhyIFAAAAAAAAAA==",
    "_self": "dbs/XYxyAA==/colls/XYxyAONhhy",
    "_etag": "\"45005090-0000-0d00-0000-613a18280000\"",
    "_attachments": "attachments/",
    "_ts": 1631197224
}
```

Listing 4: The format of charger data stored in *groupChargingMethodData*

```
{
    "id": "f7bae122-6631-46b3-af54-0c798b6b226d",
    "chargerID": "20210126-PC018",
    "carNum": "C018",
    "arrivalTime": "2021/9/9 16:24:20",
    "departureTime": "2021/9/9 18:33:30",
    "evMaximumPower": "80",
    "chargerMaximumPower": "13",
    "chargingDemand": "50",
    "currentTimeStamp": "2021/9/9 16:24:20",
    "chargerGroup": "test5",
    "status": "Preparing",
    "controlMethod": "OLCC",
    "gun1MeterValues": [
        {
            "value": "1",
            "format": "raw",
            "measurand": "ActivePower",
            "unit": "kW"
        },
        {
            "value": "10",
            "format": "raw",
            "measurand": "ReactivePower",
            "unit": "kVar"
        },
        {
            "value": "0",
            "format": "raw",
            "measurand": "ActiveConsumptionEnergy",
            "unit": "kWh"
        },
        {
            "value": "400",
            "format": "raw",
            "measurand": "DCVoltage",
            "unit": "V"
        },
        {
            "value": "60",
            "format": "raw",
            "measurand": "DCCurrent",
            "unit": "A"
        },
        {
            "value": "60",
            "format": "raw",
            "measurand": "SoC",
            "unit": ""
        }
    ],
    "_rid": "XYxyAJKqRMkTAAAAAAAAAA==",
    "_self": "dbs/XYxyAA==/colls/XYxyAJKqRMk",
    "_etag": "\"20006e84-0000-0d00-0000-613a19130000\"",
    "_attachments": "attachments/",
    "_ts": 1631197459
}
```


Listing 5: The format of charger data stored in *dynamicData*

Add/Delete/Edit/Query of the Db

After successfully establishing the database, the data in database should be able to be managed remotely from the website. Hence, it is necessary to add the Add/Delete/Edit/Query function on *"Bob"*.

The first step is to configure the environment for the Cosmos Db in MVC project. Therefore, a configuration file is created to configure the environment. The content in this file is shown in list 6. The configuration file is in Json format. The Key *"Account"* means the address of the CosmosDb; Key *"Key"* represents the password to connect to the CosmosDb; The following attributes mean the database name and the container name of the CosmosDb.

```
"CosmosDb": {
  "Account": "https://chargertestdata.documents.azure.com:443/",
  "Key": "xuu2piH4CuPEK3M2hrVcd5VS3GMdY
  2lMj7VhP45EwUDulNi2UkJVAFkdgBX0hwK9wj
  dOko1zwH20AiIzzFxBEg==",
  "DatabaseName": "testInvdata",
  "InverterData": "testdata"
},
"CosmosDbEV": {
  "Account": "https://evchargingdata.documents.azure.com:443/",
  "Key": "5vHblKBLa0xHAo1i4RUOfejOvBueq
  78GBKDu4LqH7Dfuzm633Q4jOh98V6kDttKydt
  3kVRDGJ4dnKQ0stQsMiw==",
  "DatabaseName": "EVChargingDatabase",
  "DynamicData": "dynamicData",
  "GroupChargingMethodData": "groupChargingMethodData"
}
```

Listing 6: The configuration file for CosmosDb

To connect the CosmosDb from the Website, the configuration file is not enough, the CosmosDb service need to be injected into the dependencies. Therefore, there are three functions are defined to do the service injection for three containers in two CosmosDb. For example: *InitializeInvDbAsync(IConfigurationSection configurationSection)*, the argument *configurationSection* of this function contains the Json message extracted from the configuration file, and it returns an object to the Service class which was defined in above section. For further illustration, the flow chart of how CosmosDb is connected to the MVC project is shown in fig 2.18.

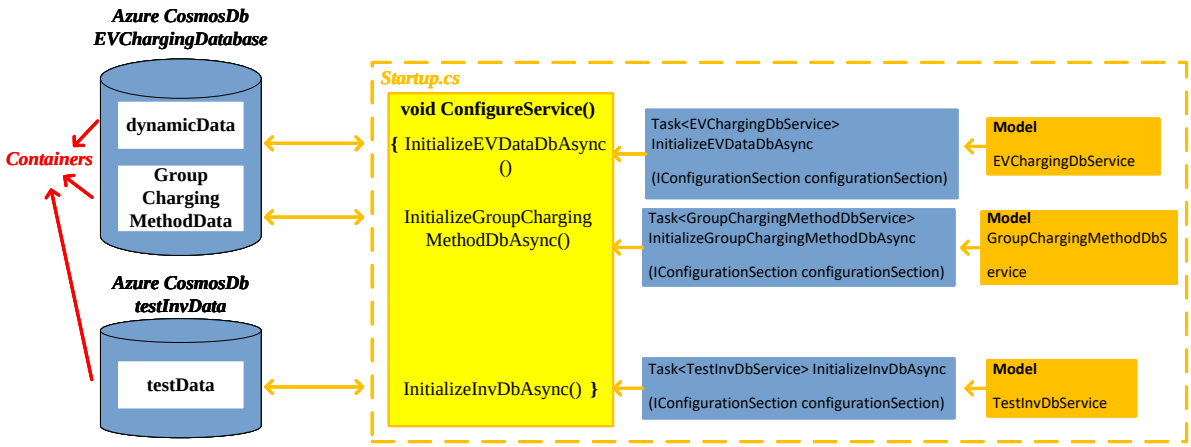


Figure 2.18: The configuration of CosmosDb

As shown in fig 2.18, In MVC project, there is a C sharp file named with startup. The function *ConfigureService()* handles the dependency service injection for the MVC project. Inside this function, there are three functions that are called, and these will functions return an object from the Model component and the CosmosDb will be connected.

Once the connection between CosmosDb and the Website is established, the items inside the CosmosDb are able to be extracted and changed by appropriate coding. Same as above, an UML class diagram will be displayed to show how Add/Delete/Edit/Query of the CosmosDb is achieved. The methods used for managing different the containers in different CosmosDbs are the same. Therefore, only the implementation of Add/Delete/Edit/Query of the container *dynamicData* in cosmosDb *EVChargingDataBase* will be presented.

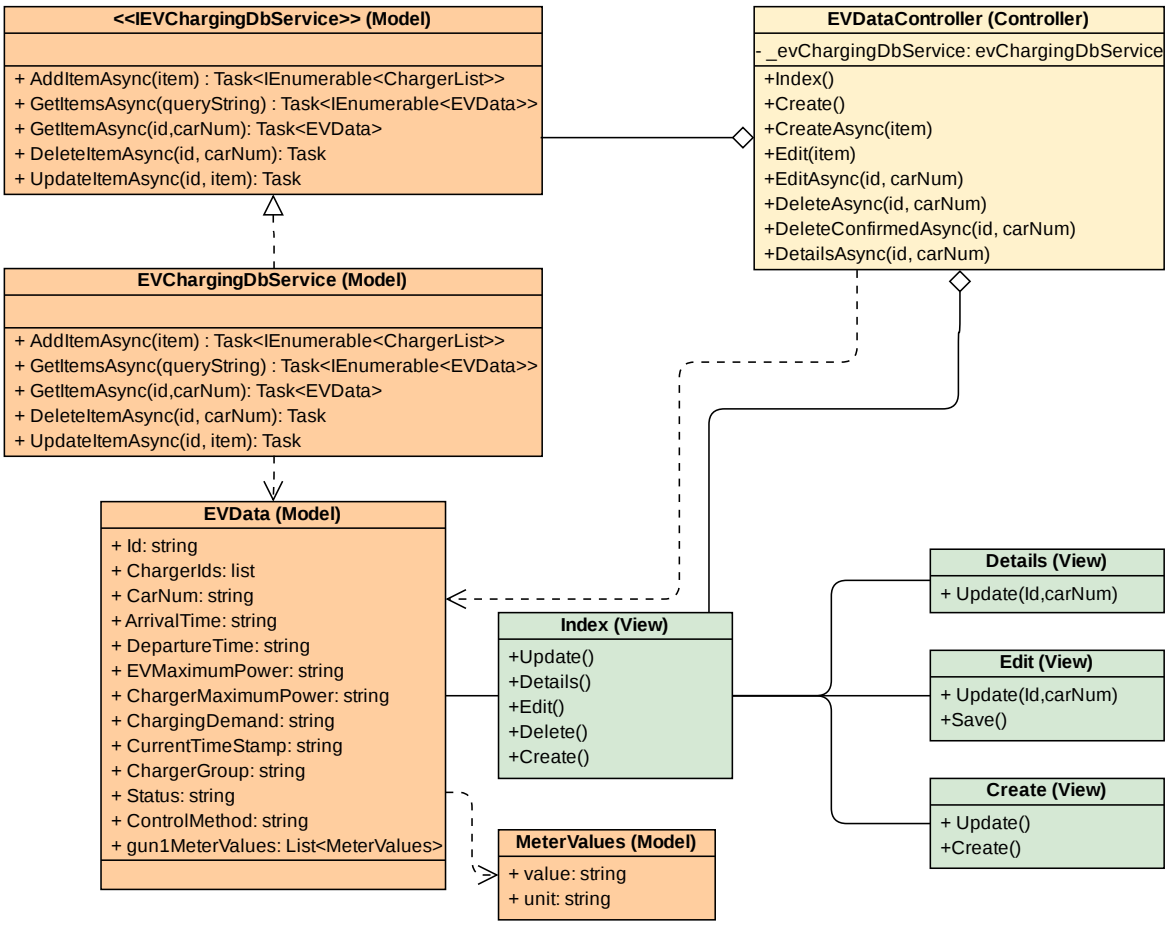
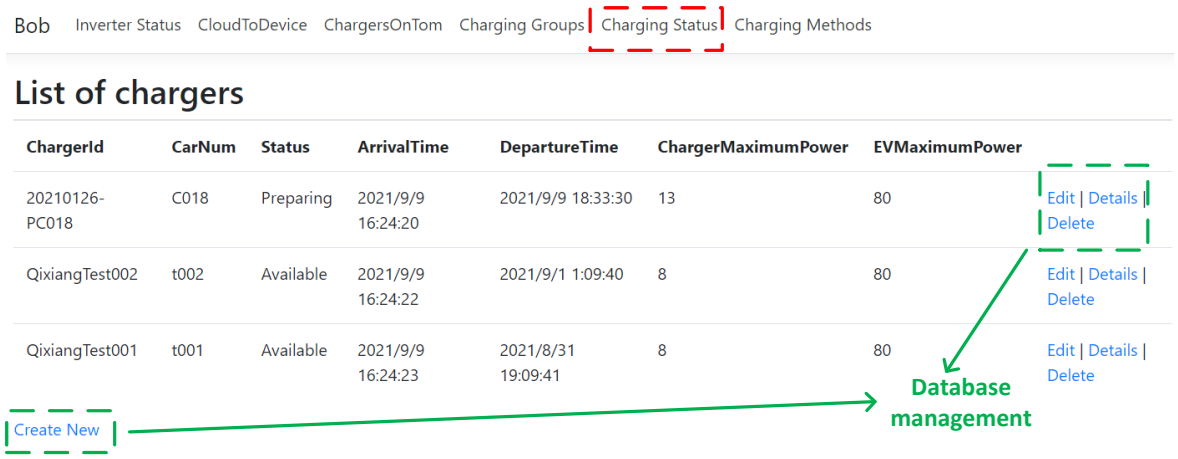
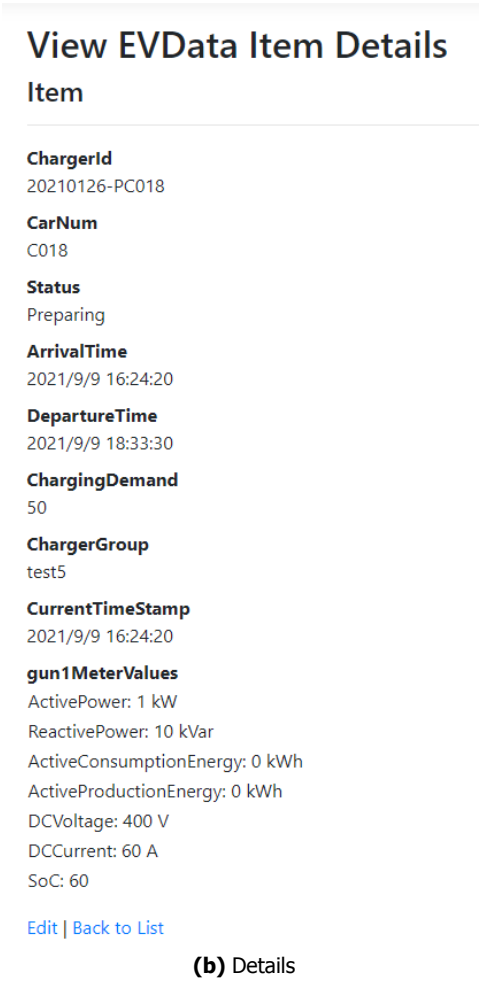


Figure 2.19: The UML class diagram of Add/Delete/Edit/Query function for the CosmosDb

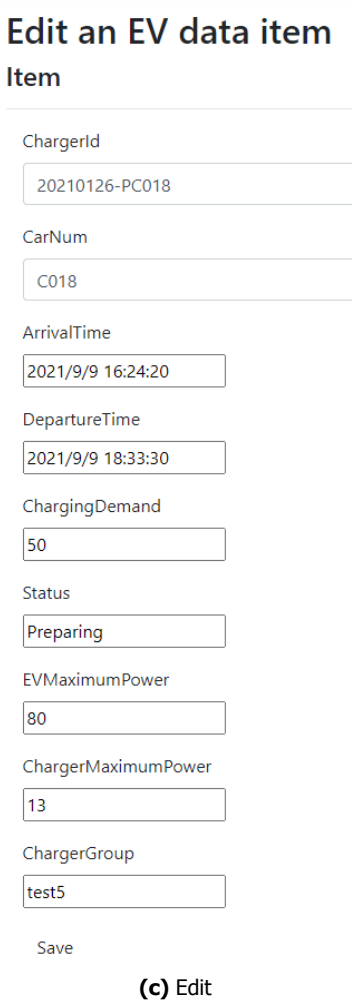
According to fig 2.19, it can be seen that the Add/Delete/Edit/Query function is implemented by the class *EVChargingDbService*. The argument *item* is an object from the class EVData. The *EVDataController* will call the function defined in *IEVChargingDbService* interface. Then the View index will display all the data read from the database. The meaning of each function is similar to the functions showed in fig 2.5. So these functions will not be explained again. The implemented view pages are shown in fig 2.20. To manage the container *dynamicData* on *"Bob"*, the *Charging Status* needs to be clicked from the title bar first. Then the list of items stored in this container will be displayed, and each item has several options for user to view/edit/delete the item in the container.



(a) Index



(b) Details



(c) Edit

Figure 2.20: CosmosDb management on "*Bob*"

2.3. Summary

In this chapter, the position of *"Tom"* and the implementation of *"Bob"* were introduced. As the summary of this chapter, there are some vital points need to be reviewed:

- The cloud platform supporting ESS-Charger system consists two parts: "*Tom*" and "*Bob*", "*Tom*" handles the remote monitoring and remote controlling of the chargers and the latter platform handles the remote power management calculating and the inverter monitoring and controlling. The information exchange between these two platforms is web APIs.
- The platform "*Bob*" was implemented under the Model-View-Controller framework, because the MVC structure is simple and clear, the development efficiency can be ensured.
- The IoT functions were implemented with the aid of Microsoft Azure IoT services.
- The database used in this project is CosmosDb which is supported by Microsoft. Since the CosmosDb is a remote database, there will be an unavoidable communication delay between the application and the database server, the more detailed communication issue caused by the delay will be discussed in Chapter4.

This Chapter only introduced how the cloud platform was designed and implemented. The detailed algorithms and how these algorithms are implemented will be presented in next Chapter. Furthermore, for the convenience of future use, all design interfaces in *"Bob"* designed to interact with the API in *"Tom"* will be summarized and listed in the following table 2.3. The first part "*Bob* Interfaces (functions) to call interfaces from *Tom*" have been tested by inputting the uri on *"Bob"* website.

Table 2.3: The web APIs from Bob

Bob Interfaces (functions) to call interfaces from *Tom*			
All request url Start with https://tpeportaldev.azurewebsites.net/api/			
Request Url	Function	Method	Body
Home/DataFromTom	Call "TpeCharger/all" interface on "*Tom*"	GET	None
Home/SearchChargerId	Call "TpeCharger/details/<chargerId>" interface on "*Tom*"	POST	string: searchString
ChargerGroup/Index	Call "TpeChargerGroup/all" interface on "*Tom*" to get the list of all charger groups	GET	NONE
ChargerGroup/CreateAsync	Call "TpeChargerGroup/Create" interface on "*Tom*" to create a new charger group	POST	chargerGroupDto: chargerGroup
ChargerGroup/EditAsync	Call "TpeChargerGroup/Update" interface on "*Tom*" to edit existing charger information	PUT	chargerGroupDto: chargerGroup

3

Charging algorithm theory & design

After constructing the cloud platform for EV charger management, the design and implementation of different charging algorithms that can be supported on this cloud platform will be introduced in this chapter. Furthermore, some simple simulated results will be shown at the end of this Chapter.

3.1. Fast charging by ratio (FCR) Method

As introudced in Chapter1, the uncoordinated charging behavior will put a heavy load on grid, and even the total charging power may exceed the upper limit of the capacity of the charging station or a house. Therefore, the first algorithm *Fast charging by ratio method* is designed to solve this problem.

3.1.1. FCR algorithm Design

Assuming there are of chargers in the charging station is n and $i \in n$, and the charging power, rated power of each charger are $P_{i_{charging}}$ and $P_{i_{max}}$ respectively. Then assume the power capacity of the charging point(group) is $P_{groupCap}$. The idea is to limit the total charging power $\sum_{i=1}^{n} P_{i_{max}}$ of chargers under the group capacity $P_{groupCap}$ without shutting down one of the charger or distributing the unbalanced power to each charger.Therefore, by finding the ratio between the specific charger's maximum power and the total charging power, this algorithm can be implemented. Say the calculated power for every chargers i in group is $P_{i_{limited}}$The equation for can be written as:

$$P_{limited} = \frac{P_{groupCap} \cdot P_{i_{max}}}{\sum_{i=1}^{n} P_{i_{max}}} \tag{3.1}$$

From the equation 4.1, it can be seen that, the maximum output power for every charger is limited by the ratio of group capacity to the sum of rated powers of chargers being charged. Then calculate the product of rated power of a specific charger and the ratio. For example, if there are two chargers with the rated power 10kw and 30kW respectively in the station, and the station's capacity is 30kW. When these two chargers charge together at the rated power, the total power will exceed the capacity. Hence, if applies the FCR method on it, the equations will be:

$$ratio = \frac{P_{groupCap}}{\sum_{i=1}^{n} P_{i_{max}}} = \frac{30}{40} = 0.75 \tag{3.2}$$

$$P_{limited_{10kW}} = ratio \cdot P_{10kW_{max}} = 7.5kW \tag{3.3}$$

$$P_{limited_{30kW}} = ratio \cdot P_{30kW_{max}} = 22.5kW \tag{3.4}$$

$$P_{total} = P_{limited_{10kW}} + P_{limited_{30kW}} = 30kW = P_{groupCap} \tag{3.5}$$

From the equations showed above, it can be seen that the total power is limited under the capacity of the charger group. The detailed implementation will be discussed in next subsection.

3.1.2. FCR algorithm implementation

As introduced in Chapter 2, all the charging algorithms will be implemented on *"Bob"* platform, the detailed implementation procedure will be illustrated step by step. The first step is to design the flow chart of how the FCR method is executed on *"Bob"*:

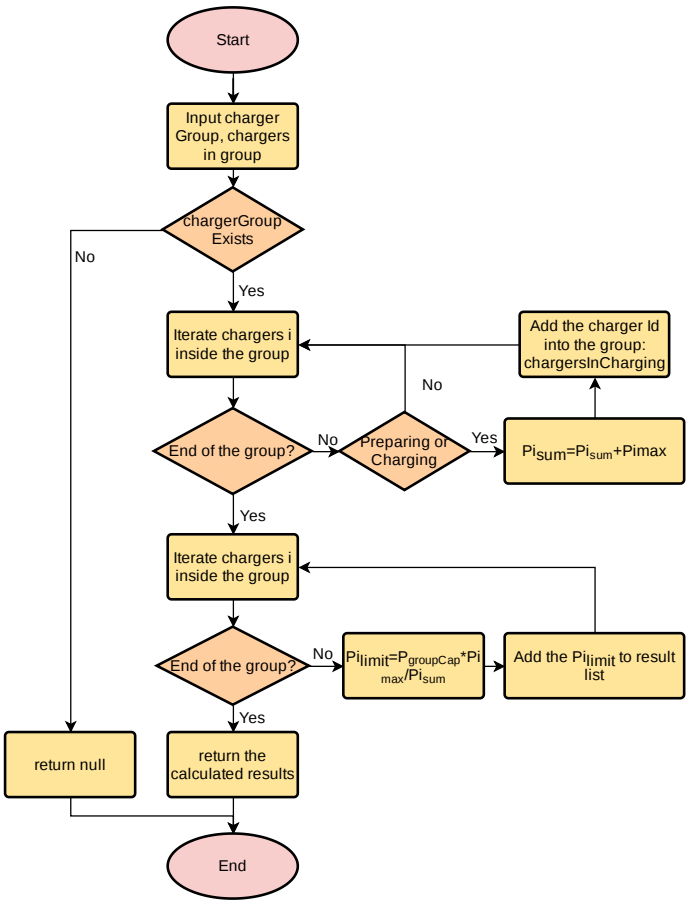


Figure 3.1: The flow chart of FRC method

Before starting the calculation, the system will put the charging group that needs to be optimized as the input. Next, the algorithm will iterate the chargers registered in this group to find out which charger is in charging (Charging) or ready to start charging (Preparing). If the charger is in charging or going to charge, the maximum output power of this charger will be recorded. Then iterate the recorded chargers and calculate the new limited power for each charger respectively. Finally, return the calculated results as a list and wait for further processing.

3.2. Coordinated Charging algorithms

In reality, the charging station owners or the community residents are interested in the operating costs.From the view of charging station owners, they want to maximize the profit between the paid to grid and the customer's charging consumption, and from the view of the community residents, they want to pay electricity bill as less as possible. Therefore, it is meaningful to propose an optimization algorithm to minimize the charging cost paid to electricity company. Thus, there is a large amount of optimization charging algorithms that has been proposed by past papers, such as [20]. However, most charging algorithms only stay at the theoretical level. To put the theoretical into practical use, there are many aspects that need to be considered. Hence, this thesis will focus on the practical implementation of the optimization algorithms instead of developing new and better algorithms. Moreover, the problems and defects encountered in the implementation of the algorithm will be discussed in the future Chapter.

In this thesis, the optimization algorithm is designed based on the algorithm in [20], also several

changes have been made to this algorithm in order to adapt to practical use cases. Before introducing the theory of the algorithm, it is necessary to formulate the problem first.

3.2.1. Problem formulation

The problem can be well formulated by introduce a scenario in reality. Assuming the community wants to build a DC fast charging station to satisfy the growing charging demands from the residents in the community. So in the community, the load characteristics can be divided into two types: elastic loads (EV chargers) and inelastic loads. The Head of the community wants to minimize the charging cost, and the residents want their charging demands of their EVs to be fulfilled within a time period that they can accept. Say the charging demand of an EV i is $D_i(kWh)$ and suppose the time of charging starts, the time of charging ends, the rated power of charger are t_{i_s}, t_{i_e} and $P_{i_{rated}}$ respectively. The first equation can be written as:

$$D_i \leq P_{i_{rated}} \cdot (t_{i_e} - t_{i_s}) \tag{3.6}$$

Subsequently, to derive the minimum charging cost, the calculation method of electricity price needs to be modeled, according to [20–22], the electricity price can be written as a linear function which the instant price ($C_{instant}$) is relevant to total instant load at the moment:

$$C_{instant} = a + 2b \cdot z_t \tag{3.7}$$

Where a and b are varying by time the exact value depends on the quoted price provided by electricity company. z_t represents the total instant load at the time t. Furthermore, as mentioned at the beginning of this section, there are two types of loads, say the total elastic load caused by EV chargers is $L_{charger}$, and the instant power for each charger is x_{i_t}. Then assume the total inelastic load caused by other purposes is L_{other}. Then, the total loads at the time t L_{tot} can be replaced by:

$$L_{tot} = L_{charger} + L_{other} = \sum_{i=1}^{N} x_{i_t} + L_{other} \tag{3.8}$$

As for equation 3.7, this equation only decides the total instant price, to get the actual cost of the charging behavior, the equation 3.7 can be further derived as:

$$C_{charging_{instant}} = \int_{L_{other}}^{L_{tot}} (C_{instant}(z_t))dz_t = \int_{L_{other}}^{L_{tot_t}} (a + 2bz_t)dz_t \tag{3.9}$$

Thus, base on the equation 3.9, to get the actual charging cost over a fixed time period T, a time integral to equation 3.10 will be applied, the charging cost can be derived:

$$C_{charging} = \int_0^T \int_{L_{tot}}^{L_{other}} (a+2bz_t)dz_t dt = \int_0^T (a(\sum_{i=1}^{N} x_{i_t}+L_{other})+b(\sum_{i=1}^{N} x_{i_t}+L_{other})^2-(aL_{other}+bL_{other}^2))dt \tag{3.10}$$

Hence, according to the equation 3.10, the objective function of this problem can be found. The optimization variable is x_{i_t} and the objective is to have minimum cost $C_{charging}$ by calculating the value of x_{i_t} for each vehicles which are in charging status. Besides, while meeting the minimum charging cost, it also needs to meet the user's charging demands and keep the charging power within an allowable range. Thereby, the objective equation and the constraints can be expressed as follows:

$$\min_{C_{charging}(x_{i_t})} = \int_0^T (a(\sum_{i=1}^{N} x_{i_t} + L_{other}) + b(\sum_{i=1}^{N} x_{i_t} + L_{other})^2 - (aL_{other} + bL_{other}^2))dt \tag{3.11}$$

$$s.t. \quad D_i = \int_{t_{i_s}}^{t_{i_e}} x_{i_t}, i \in [1, N] \tag{3.12}$$

$$0 \leq x_{i_t} \leq P_{i_{rated}}, i \in [1, N], t \in [t_{i_s}, t_{i_e}] \tag{3.13}$$

$$0 \leq sum_{i=1}^{N} x_{i_t} \leq P_{groupCap} i \in [1, N] \tag{3.14}$$

The $P_{groupCap}$ in constraints 3.18 represents the maximum capacity of the charging station in community. By solving quadratic optimization above, the minimum cost can be found. Nevertheless, the integral is in continuous format, to put the algorithm into practical use, the continuous integral needs to be discretized. Besides, in reality, it is not possible to know or predict every EV's arrival time so accurately. Therefore, before going further, a simple offline charging algorithm is proposed.

3.2.2. Offline Charging algorithm (OFFCC)

In this section, the discretize of equation 3.15 will be discussed, and because this algorithm is offline, all the other variables will be treated as known. By referring [20], the continuous time interval can be discretized by events, the length of each event is different. Fig 3.2 is displayed as an example for better illustration.

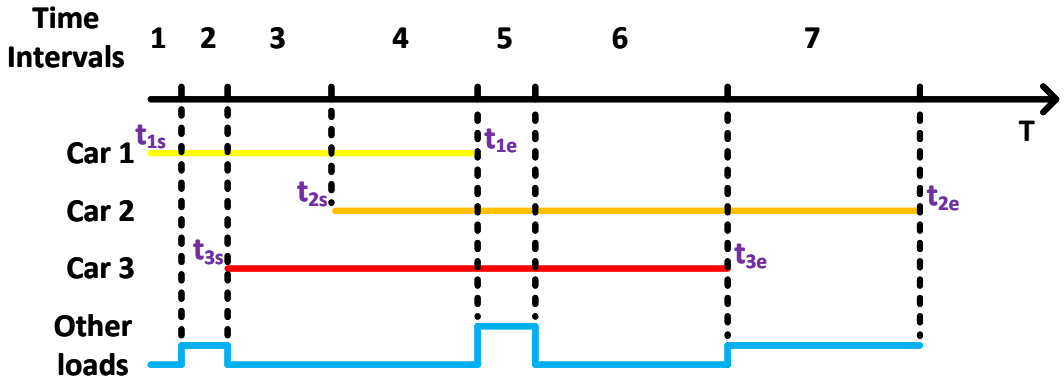


Figure 3.2: The discretization exmaple for OFFCC

From the figure, it can be seen that there are 7 intervals over a fixed time period. The intervals are divided by events, an event could be the arrival or departure of an EV, or the change of the other loads. During a specific time interval, the charging power and the amount of other load will be fixed, and there will be an optimal solution to charging power for an interval, the solution will remain unchanged until the next interval is reached. The proof of this conclusion can be found in [20]. Based on this assumption, say there will be k intervals in the future and the time length for each interval will be $\Delta T_k = T_{event_t} - T_{event_{t-1}}$. By doing so, the continuous problem can be discretized as follows:

$$\min_{C_{charging}(x_{i_k})} = \sum_{k=1}^{k} (a(\sum_{i=1}^{N} x_{i_k} + L_{other}) + b(\sum_{i=1}^{N} x_{i_k} + L_{other})^2 - (aL_{other} + bL_{other}^2))\Delta T_k \tag{3.15}$$

$$s.t. \quad D_i = \sum_{i=1}^{k} x_{i_k} \Delta T_k, i \in [1, N] \tag{3.16}$$

$$0 \leq x_{i_k} \leq P_{i_{rated}}, i \in [1, N] \tag{3.17}$$

$$0 \leq sum_{i=1}^{N} x_{i_k} \leq P_{groupCap} i \in [1, N] \tag{3.18}$$

Furthermore, for this offline algorithm, assuming that the charging schedule for the next day is known, and the load profile is known, the only unknown value of this equation set is x_{i_k}. Therefore, the optimization problem can be solved. However, in practical, the arrival time of the EV is unknown, and the load profile is hard to be predict. So, for online charging algorithm, some changes has to be made based on the OFFCC algorithm.

3.2.3. Online Charging algorithm (OLCC)

Since the future event cannot be predicted, the OLCC will recalculate the result every time an event occurs by assuming there will be no more event after this moment. Once an EV arrives, the system will let the user input the charging demand and the departure time. Once the charging procedure is initiated, the system will record the starting time and do the calculation. So, compare with the OFFCC, though the equations for these two methods are same, the OLCC method requires multiple compute to make sure the online optimization results as closer as possible to the optimal value.

Thus, if an event occurs, all the data at the time have to be updated again, especially under the charging demand. According to the status of the charger, the charging demand at the time tD_{i_t}can be divided into three conditions:

$$D_{i_k}\begin{cases} 0 & Chargingfinished \\ D_i & Chargingstarted \\ D_{i_k} - x_{i_{k-1}} \cdot (\Delta T_{k-1}) & InCharging \end{cases} \tag{3.19}$$

Equation 3.19 shows different cases of charging demand. Besides, the event interval and the length of the interval also needs to be updated. In addition, same as the OFFCC, the online algorithm cannot predict the future as well. So, to make sure all the charging demand can be satisfied a acceleration factor Q is introduced [20]. Suppose the optimal value for charging power is x^*_{ik}, the acceleration factor Q will be used as a multiplier and multiplied by x^*_{it} to get the final charging power $\hat{x_{ik}}$, the detailed equation can be expressed as follows:

$$Sum_k = minQ \cdot \sum_{i=1}^{N} x_{i_k}, \sum_{i=1}^{N} P_{i_{rated}} \tag{3.20}$$

$$\hat{x_{ik}} = min(x_{i_k} + \frac{P_{i_{rated}} - x_{i_k}}{\sum_{i=1}^{N}(P_{i_{rated}} - x_{i_k})} \cdot \frac{Q-1}{Q} Sum_k, P_{i_{rated}}) \tag{3.21}$$

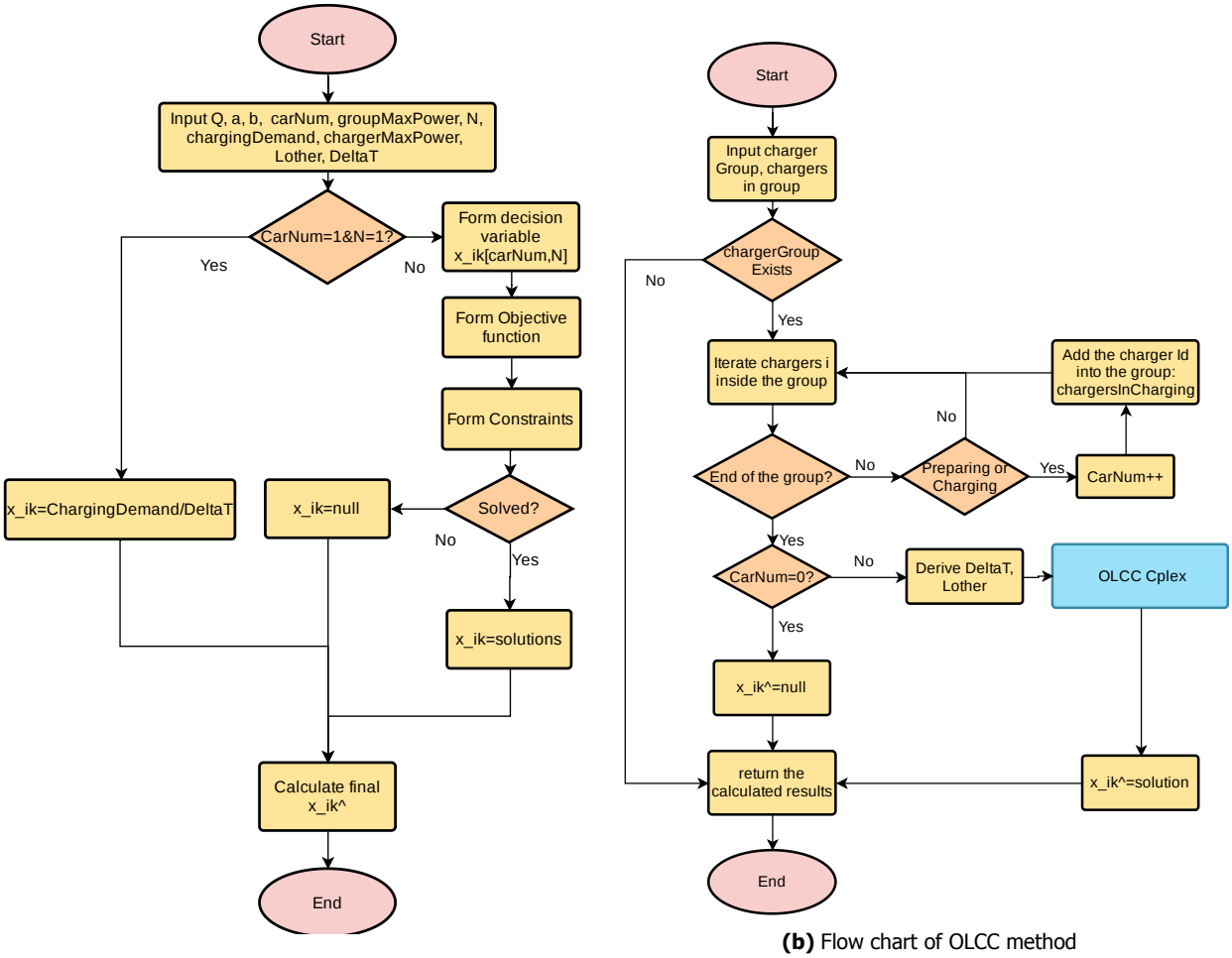
The $\hat{x_{ik}}$ derived from equation 3.21 will be the final charging power that going to be sent to the charger, by doing so, although the charging cost will be slightly increased, the charging demand can be satisfied. Subsequently, with the equations derived above, the OLCC algorithm can be achieved, the processing steps can be divided into following steps:

- Start the controller, and the number of interval will be set to 0.
- Once an event is occurred, such as: EV arrival, departure or other load changes, the controller will be triggered, and a new interval will be generated, the current time will be the start point of the new interval. At the meanwhile, all the parameters will be updated and input as the input to the controller as well. The parameters are: $P_{i_{rated}}$, t_{i_s}, t_{i_e},D_{i_k},L_{other},ΔT_k.
- Solve the quadratic problem described in equation 3.15 to get the optimal solution x^*_{ik} for each charger in charging.
- Calculate the final value of charger power $\hat{x_{ik}}$ by applying the equation 3.21 to x^*_{ik}.
- Return to step two and ready for the next event to be triggered.

For better illustration, the above steps can be concluded into two flow charts in fig 3.3a and fig 3.3

With the aid of flow chart fig 3.3a, the OLCC algorithm can be implemented by using the CPLEX in *"Bob"*. However, from the fig 3.3a, it can be seen that there are multiple inputs, and some of inputs have to be derived from the information collected from chargers. Therefore, to derive from the input and make the OLCC algorithm functional in practical, a larger function is defined, and its flow chart is shown in fig 3.3b, the blue block represents the function which is shown in fig 3.3a. The final calculated power of each charger will be formed in a list and with a name corresponding to the power.

Furthermore, in reality, the operator will choose different charging algorithm for different charging stations to fit for different requirement and situation and the system also need a trigger to trigger the controller. Therefore, a master controller is needed for "*Bob*". The more detailed design and the implementation about this controller will be introduced in next section.



(a) The flow chart of solving the OLCC method with CPLEX

(b) Flow chart of OLCC method

Figure 3.3: The implementation of OLCC method

3.3. "PowerConfig" API on *Bob*

As discussed in the end of the last section, a master controller is required for advanced management. This controller can be achieved by designing an external interface for third party application to call such as *"Tom"*, and the name of this controller is *PowerLimitController*. Inside this controller, a POST interface is defined, and the route of this interface is defined as: *"https://<ip address>/api/powerconfig"*, the function coordinate to this route is *SetPowerTrigger(ChargerStatusDto changedCharger)*. Once, this interface is called by other application, the algorithms defined on "*Bob*" will be triggered.

In this project, this interface will be used by "*Tom*". To trigger the algorithm, the message sequence can be concluded into fig 3.4. By referring the fig 3.4 and the flow charts above, the operation mechanism of the algorithm can be well explained:

- First, when the system is running, and if there is a status change of the charger, the software in charger will POST a charger change information to *"Tom"*. There are four status in total: *Available*, means the charger is online and awaits for charging; *Preparing*, means the charger has been plugged into the EV and ready for charging; *Charging*, means the charger is charging an EV; *Finishing*, means the charging process has ended but the plug haven't been removed from the EV.
- Then, the "*Tom*" will process this message from the charger, convert the message format in order to POST it to *"Bob"* through the interface *api/PowerConfig*.
- Next, the interface on "*Bob*" will call coordinate function *SetPowerTrigger* to do message process-

ing and calcuation.

- Subsequently, during executing the *SetPowerTrigger* function, the calculation result will be sent to "*Tom*" through interface *SetPower* on *"Tom"*, once the "*Tom*" received the *SetPower* command, the command will be sent to EV chargers. By doing so, the control loop can be enclosed.
- Finally, at the end of the *SetPowerTrigger* function, a response will be returned to "*Tom*" to clarify if the POST request is success.

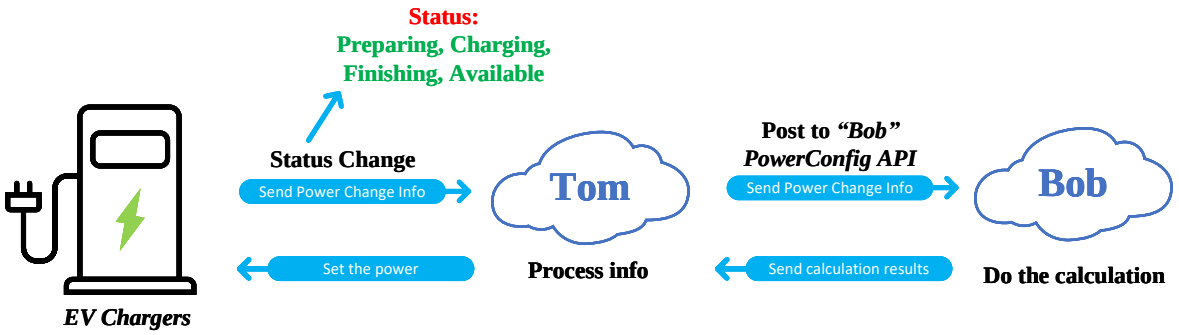


Figure 3.4: The message sequence flow from the charger to cloud platform

Above paragraphs introduced how charging algorithm is triggered in a macro way, to further elaborate how function *SetPowerTrigger* is functioned, a flow chart is made and shown below in fig 3.5. The general idea is to use the single changed charger data to trigger the algorithm, and algorithm need to gather all the information required by the algorithms, all the charger data which includes real time and past should be passed into this function. To do that, in flow chart, the first step is to locate the charger group and find out all charger Ids inside this group. Next get each chargers' real time data on *"Tom"* by using the function *GetChargerDetials* defined in 2.2.2. Then, as mentioned in section 2.1, *"Tom"* platform only provides the real time data to external applications. Thus, the past data has to be stored in database of *"Bob"* which was introduced in section 2.2.4. In addition, the database on "*Bob*" also record the charging algorithm for each charger groups, so by querying to the database, the *PowerDispatch* controller will know which optimziation algorithm is chose for target charger group .Once the real time and past data has been passed and updated to the function and database, the optimization algorithm is specified, the algorithm can be started. The *PowerDispatch* controller will select proper algorithm based on the specified name of charging algorithm (In this project, the OLCC and FRC method can be selected). The calculation results will be formed in a list. Afterwards, if the results list is not empty, the function will iterate the items inside the list to get the target power for each chargers and send the power setting command to "*Tom*" and the OK message will be the response to this POST action. On the contrary, if the results list is empty, which means no chargers need to be controlled or there is a fault during calculation, no command will be sent to *"Tom"* and the bad request will be sent as the response to this POST action. Finally, if the *"Tom"* received the power setting command, the *"Tom"* will forward this message to specific charger to limit its maximum charging power.

Moreover, in practical use, to ensure the power setting command can be sent and processed before the charging starts, the algorithm will be initiated at the *Preparing* status, which means if the customer plug the charger into the EV, the charger will be turned from *Available* to *Preparing*. If the algorithm found it is in *Preparing* status, the algorithm will be applied to this charger. Therefore, the power of this charger can be limited before customer press the start charging button on screen.

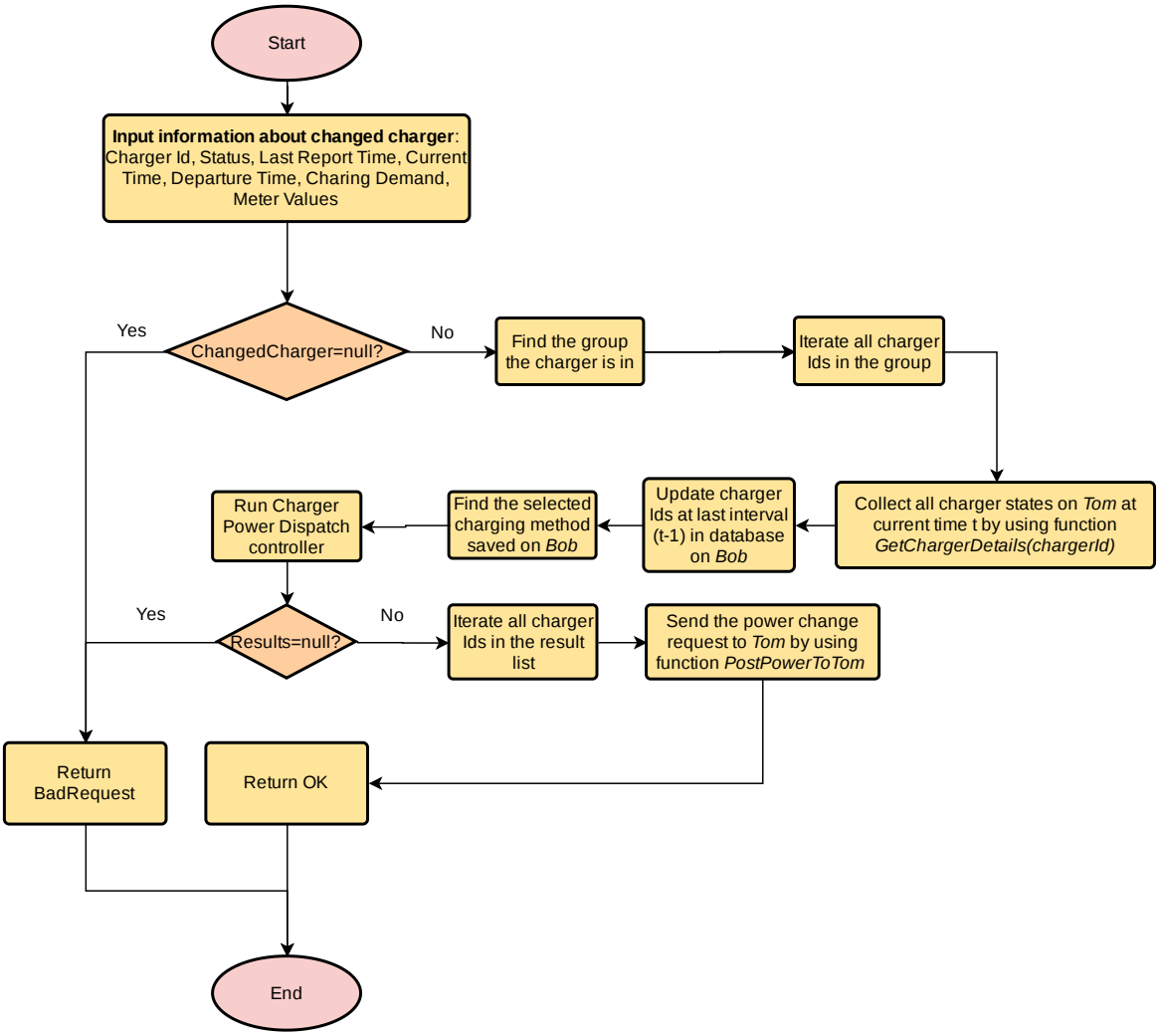


Figure 3.5: The flow chart of how SetPowerTrigger function functioning

3.4. ESS management

As mentioned in Chapter 1, the ESS will be contained in this system. Therefore, a proper ESS management has to be designed and implemented in this system. Nevertheless, due to development objective and experiment equipment, only a simple peak-shaving method which will be used to shave the peak power or provide power compensation to grid will be proposed in this section. The complicated algorithm with SOC management will not be considered. The proposed peak-shaving method can be simply concluded in equation 3.22

$$\begin{cases} P_{ESS} = P_{grid} - P_{dischargingPoint} & ESS \quad Discharging \quad when \quad P_{peakshavingPoint} \leq P_{grid} \\ P_{ESS} = P_{chargingPoint} & ESS \quad Charging \quad when \quad P_{grid} \leq P_{chargingPoint} \end{cases} \tag{3.22}$$

Furthermore, this peak-shaving method has two modes for user to choose: manual and automatic.

- Manual: this method only enables ESS charging or discharging to the grid with a designated time zone and fixed power. The charging/discharging time zone and charging/discharging power is set by the user.
- Automatic: this method will do the peak shaving automatically alone with the charging optimization algorithms. The user only able to set the charging/discharging point for ESS.

For example, fig 3.8 shows the options for user to set on *"Bob"*. It can be seen that, the *"Bob"* platform records the selected charging algorithm, ESS management method, charging/discharging

start point, and basic setting options for peak-shaving manual. In this figure, the selected charging optimization for group *Test5* is OLCC, and the ESS method is Peak-shaving automatic, the charging start point is 0.5kW, and the discharging start point is 10.5kW.

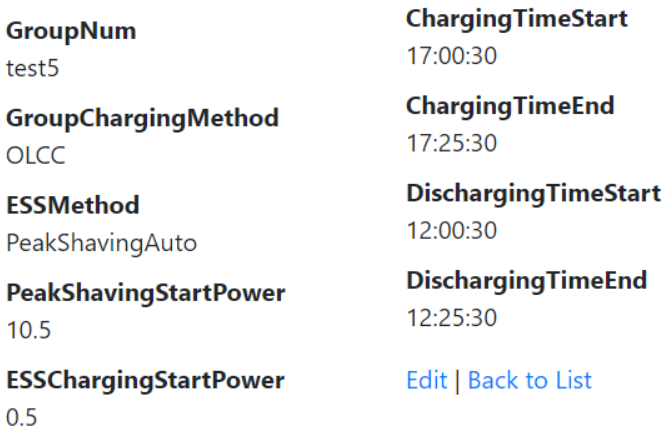


Figure 3.6: The ESS managment options on *Bob*

To implement the peak-shaving auto method in order to coordinated with the charging optimization algorithm, the more functions can be added into the *SetPowerTrigger* function. Hence the flow chart of *SetPowerTrigger* in fig 3.5 can be further modified, which is shown in fig 3.7. The newly added ESS management blocks are highlighted in a blue color. It can be seen that the peak shaving method is fully decoupled with the charging optimization algorithm. Therefore, the whole power management system *"Bob"* is both adapted to the system with or without the ESS.

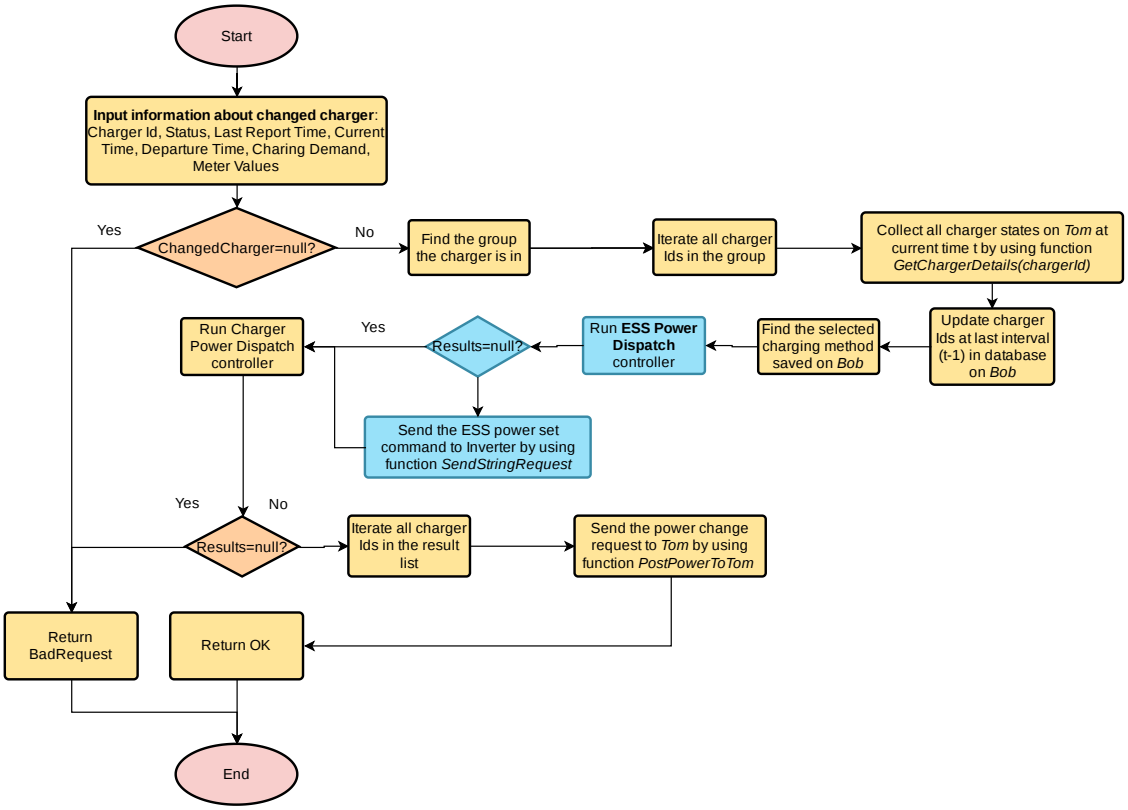


Figure 3.7: The flow chart of how SetPowerTrigger function functioning with ESS management

3.5. Simulation

Above sections have introduced and discussed the theory and the implementation of the charging and ESS management algorithms proposed in this thesis. Before putting these algorithms into practical use, the simulation must be done. Therefore, in this section, the performance and of the proposed charging and ESS management algorithms will be evaluated through the simulation.

3.5.1. Simulation parameters

There are only two methods proposed in this project FRC and OLCC. However, the simulation is easier to implement the other algorithms. Hence, to highlight the performance difference between different algorithms. The other algorithms also have been made and added to the simulation, such as OFFCC and average charging (AVG). The OFFCC is the foundation of the method OLCC, it aslo can be called planned charging. The AVG algorithm is a simply averaging method which will average the charging power according to the charging demand: $P_{AVG} = \frac{D_i}{t_{i_e} - t_{i_s}}$.

In addition, as for OLCC, the electricity price has to be known before starting the simulation. In this simulation, based on [22], the value of a and b are defined as: $a = 10^{-4}€/kWh$, $b = 0.6 \cdot 10^{-4}€/kWh/kW$. Therefore, the electricity price from equation3.7 can be expressed by $C_{instant} = 10^{-4}€/kWh + 2 \cdot 0.6 \cdot 10^{-4}€/kWh/kW \cdot z_t$. Then, there are only two types DC chargers with two different maximum charging power level will be considered in this simulation and they are 30kW and 20kW respectively. As for charging demand for each vehicle, the charging demand are divided into three levels: short, medium and long term, the values are 35kWh, 45kWh, 75kWh respectively. The simulation are divided into three conditions, half-day simulation with light traffic, all day simulation with medium traffic and all day simulation with heavy traffic. In order to simplify the simulation process, the minimum time interval are set to 0.5h. The arrival and departure time for each EV will be listed in tables below, and with the aid [23, 24], the peak arriving hours are defined from 14:00-16:00 and 6:00-8:00. In addition, the OLCC introduced a speed factor Q, the Q is set to 1.1 [20]in order to reach the best performance of the OLCC algorithm. The EV charging data and simulation results will be shown in next section.

3.5.2. Simulation results

At the beginning the most simple case will be simulated, which is half-day light traffic. In this simulation, the capacity of the charging station is set to 50 kW, the time starts from 12:00 to 24:00, the specific EV charging timetable and load profile are shown in table 3.1 and table 3.2 (the load will remains unchanged reaching the next time point), and the simulation results and the cost of different methods will be shown in fig 3.8,fig 3.9 and fig 3.10.

Table 3.1: The EV timetable for short-term simulation

EV Num	Time of Arrival t_{i_s}	Time of Departure t_{i_e}	Charging Demand $D_i \quad (kWh)$	Max Charging Power $P_{i_{rated}} \quad (kW)$
1	12:00	16:00	45	30
2	14:00	20:00	35	30
3	14:00	24:00	75	20
4	15:00	18:00	35	30
5	16:30	20:00	45	20
6	18:00	22:00	45	20
7	20:00	24:00	45	30

Table 3.2: The EV timetable for short-term simulation

Time	12:00	13:00	14:00	15:00	16:00	17:00	18:30	19:00	20:30	21:00
Load (kW)	30	20	16	24	36	48	60	50	40	30

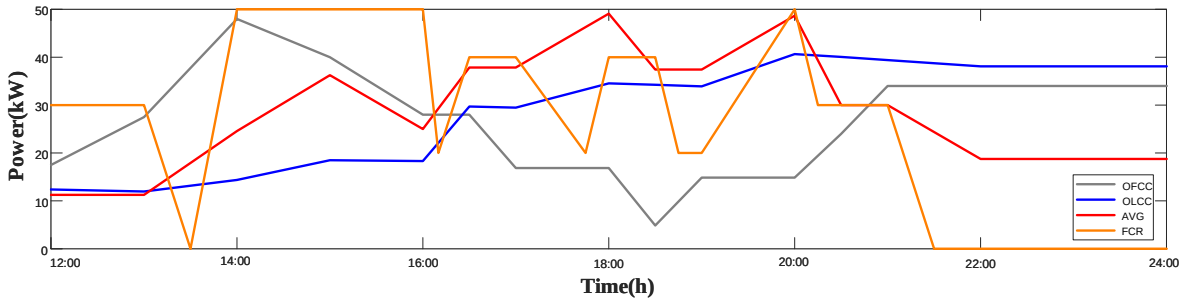
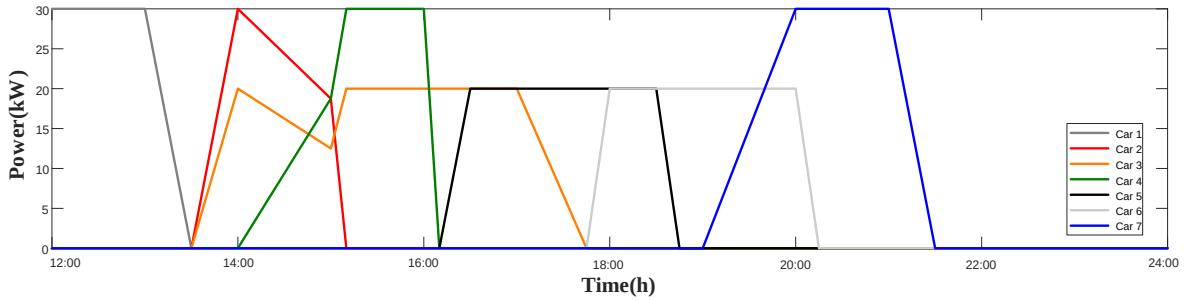
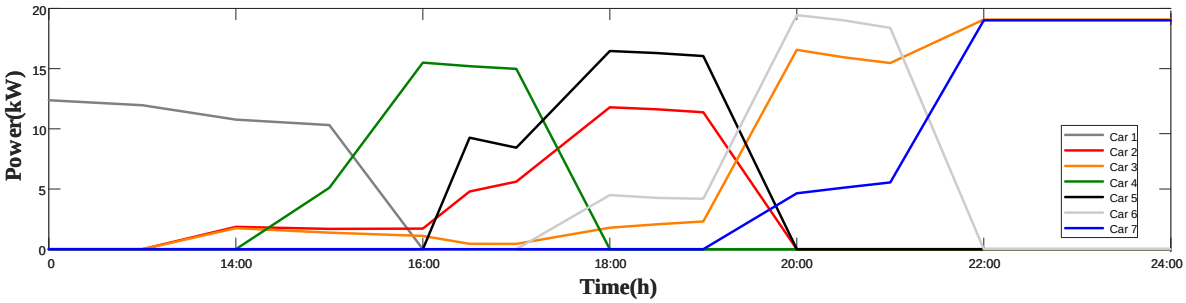


Figure 3.8: The total charging power with different methods from 12:00 to 24:00 for a day under light traffic



(a) Charging power for each EV with FCR method



(b) Charging power for each EV with OLCC method

Figure 3.9: Charging power for each EV with different methods in short-term simulation

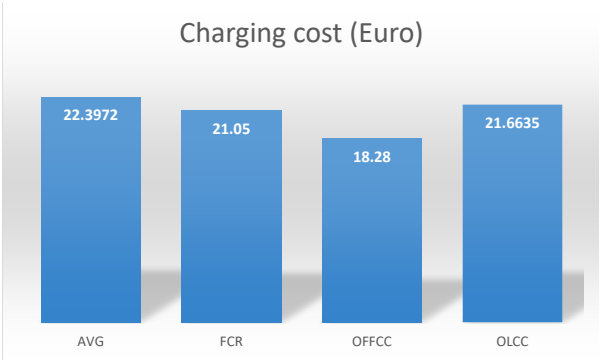


Figure 3.10: The total cost of different methods for short-term simulation

From the results, it can be seen that the cost of OFFCC method is lowest, it is because that the all the charging timetable for this time duration is known, the result must be optimal. As for online algorithms, the FCR methods spends less money than OLCC method. This may be caused by the speed factor in OLCC under low traffic condition. Therefore, more simulation will be done under higher traffic condition.

Similarly the simulation are done under the medium traffic and heavy traffic condition, the simulation data and the results under medium will be shown in following tables and figures, the station capacity is set to 90kW. The results for heavy traffic condition were put in Appendix C.

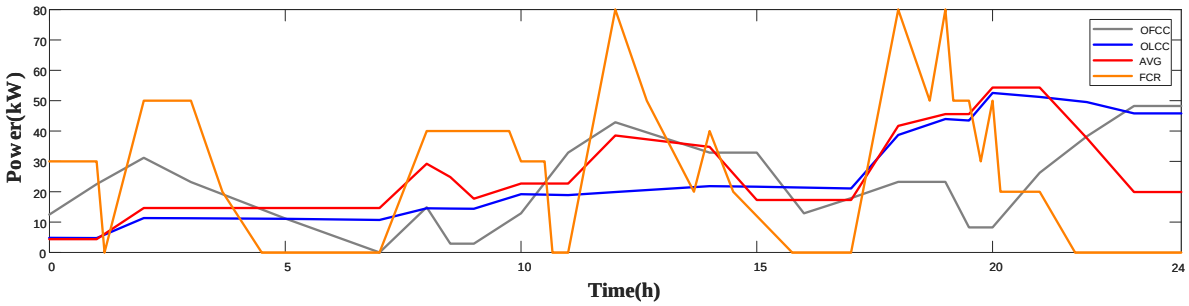
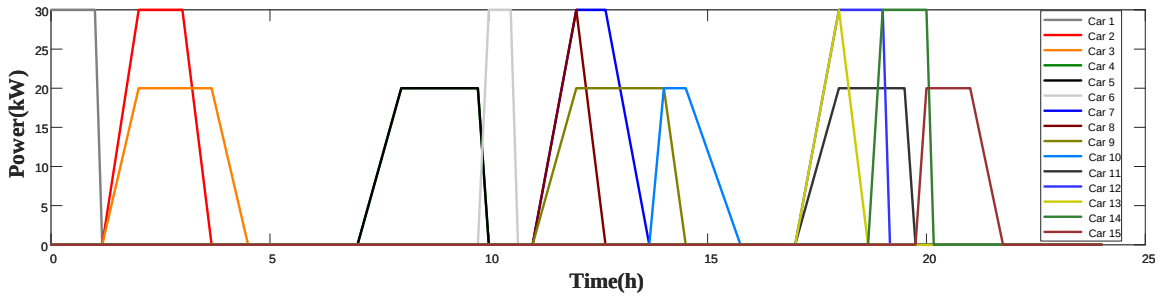
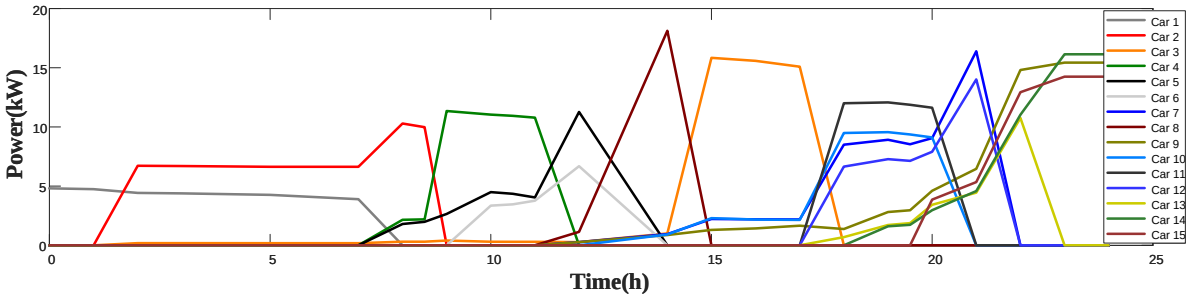


Figure 3.11: The total charging power with different methods for a day under medium traffic



(a) Charging power for each EV with FCR method



(b) Charging power for each EV with OLCC method

Figure 3.12: Charging power for each EV with different methods in medium-term simulation

Fig 3.11, 3.12 shows the power dynamic of whole station or for single EV by applying different charging methods, and the fig 3.13 shows the cost with different methods under medium traffic condition. Moreover, heavy traffic condition in appendix C shows the same result, which is the OLCC method will further reduce cost while multiple EVs are in charging at the same time. Especially under heavier traffic condition. Nevertheless, the FCR shows better performance under light traffic condition. Therefore, the FCR can be used for charger stations which contains less than 5 chargers, and OLCC is suitable for larger charger station. Further discussion will be given in next chapter.

In addition, the ESS management also have been simulated in a simple way: Assume there is only

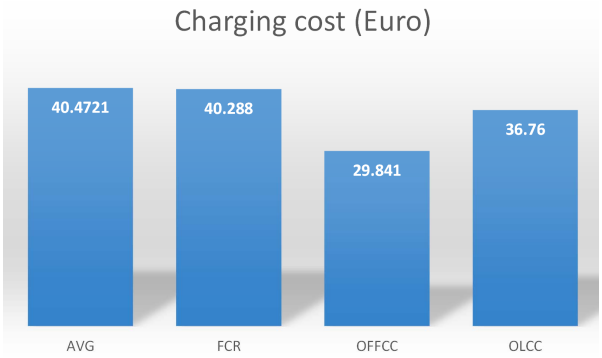


Figure 3.13: The total cost of different methods for medium-term simulation

one charger is in charging and no charging algorithms will be applied on this charger. The maximum charging power of the charger is 10kW, and the peak-shaving starting point for ESS is set to 8kW, the ESS is simulated by OpenEMS [25]. The result is shown in fig 3.14. In this simulation shows the charging power is shaved 2 kW by ESS and therefore the grid power is limited around the 8kW which is below the peak-shaving starting point.

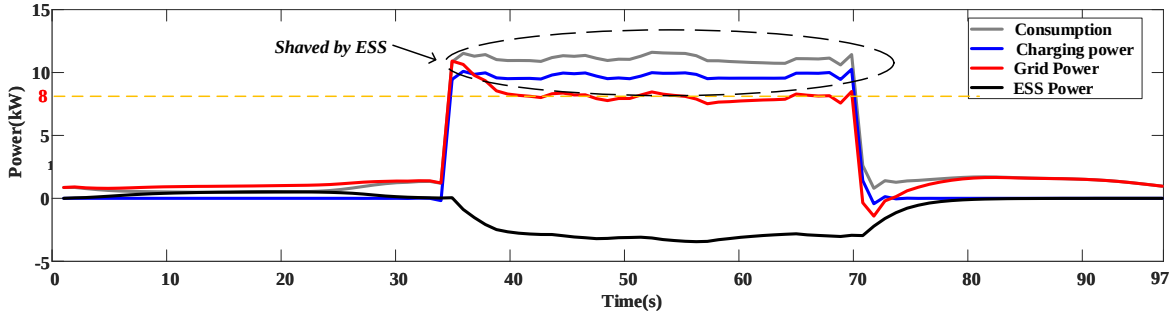


Figure 3.14: The simulation result of the Peak shaving algorithm

3.6. Summary

In this chapter, the theory and the implementation of charging and ESS algorithms were discussed. The simulation results were displayed and briefly analysed. As the summary of this chapter, there are some vital points need to be reviewed:

- Two charging algorithms were proposed in this chapter: Fast charging by ratio (FCR) and Oline Coordinated Charging (OLCC) algorithm. The FCR methods is fit for small charging stations or the light traffic condition, and the latter is more suitable for the large scale charging station which under heavier traffic condition.
- The ESS managment method was implemented as well, through the peak shaving, the charging station will gain more power capacity to keep all the chargers charging in a high power rate.
- All the calculation were finished on cloud.
- The cloud control needs to be triggered by the external interface *"PowerConfig"* on *"Bob"*, then the *PowerDispatchController* will do the rest processing, such as selecting the correct charging method, recording the past data, and initiate the calculation, get the calculation result and send the proper instructions to *"Tom"* or the inverter.

Similarly, the external interface *PowerConfig* will be summarized in table 3.3, and the message format can be found in Appendix B. Moreover, this "*Bob* external Interfaces" have been tested and verified in POSTMAN.

Table 3.3: The web APIs from Bob

Bob external Interfaces

API Name	Function	Method	Body
api/powerConfig	Post charging power to a charger through "Charger/setpower/<chargerId>" provided by "Tom"	POST	ChargerStatusDto: changedCharger

4

Experiment validation & evaluation

Chapter 2 introduced the implementation of the *"Bob"* platform and Chapter 3 illustrated how the algorithms is designed and implemented on *"Bob"*. Hence, the performance of whole platform and algorithms need to be validated and evaluated through the practical test.

In this chapter, all the device parameters and how their were connected will be introduced. Furthermore, the algorithms will be tested through a real charger and real ESS. The system overview is shown in fig 4.1. More details will be explained in following sections. Finally the results will be analysed and discussed at the end of this chapter.

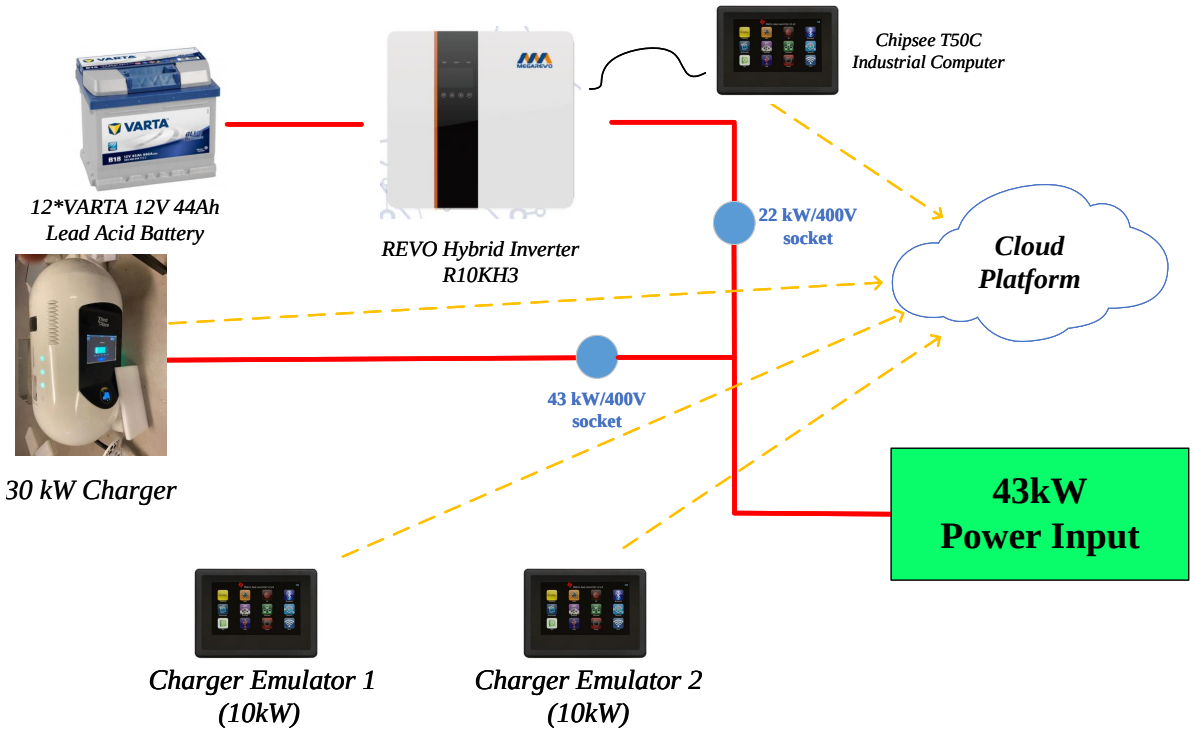


Figure 4.1: Hardware connection schematic

4.1. Experiment devices

According to figure 4.1, the whole system has 6 components and two sockets. They are two Charger emulators, a real 30kW DC charger, a battery pack, an inverter and a Industrial computer which will be used to control the inverter. Besides the inverter and the battery pack, other components are connected to cloud. Furthermore, the $30kW$ DC charger is plugged into $43kW/400V$ socket and the inverter is plugged into $22kW/$ socket. The other industrial computers are supported by a 24V DC source respectively. The total capacity of the laboratory is 43kW. The parameters of these components will be introduced in following subsections.

4.1.1. Chargers

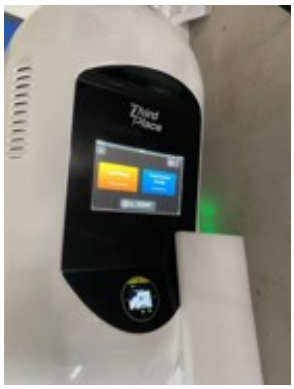
There are three chargers that will be used for experiment, one is a 30kW DC charger which provided by Third Place Energy B.V. The other two chargers are virtual chargers which are simulated by two industrial computers. The parameter of the DC charger can be found in table 4.1, and the industrial computers is provided by Chipsee. Ltd. The data for this industrial computer is concluded in table 4.2. The photo of the DC charger and the charger emulator are shown in fig 4.2. In addition, an EV from Skoda is used for 30kW DC charger, which can be seen in fig 4.2c.

Table 4.1: The parameters of 30kW DC Charger

Charger Name	Input Voltage	Output Voltage	Maximum Input Current	Maximum output Current	Power Factor
Power Capsule 30kW	400±15% AC	200-950V DC	50 A	100 A	≥0.99

Table 4.2: The parameters of Industrial Computer

OS	CPU	DC Input	Power Consumption	Size
Linux	ARM Cortex A8	6-36 V DC	3.5 W	10.5 inch



(a) The photo of 30kW DC charger

(b) The photo of two charger emulators

(c) The photo of EV for DC charger

Figure 4.2: The photo of three chargers in lab

4.1.2. ESS

As for ESS, the parameters of battery pack and the inverter are shown in table 4.3 and 4.4, the manufacturer of the inverter is MEGAREVO, the type is 10kW hybrid inverter.The battery branch is VARTA. The photo of the ESS can be found in fig 4.3.

Table 4.3: The parameters of Hybrid Inverter

Inverter Name	Maximum Power	Rated Output Voltage	Maximum Input Current	Maximum output Current	Power Factor
R10KH3	10 kW	400V AC	15.8 A	33.4 A	≥0.99

Table 4.4: The parameters of Battery

Battery Type	Rated Voltage	Capacity	Amount	total output power
Lead-Acid	12 V	44 Ah	12	144 V

(a) The photo of Inverter

(b) The photo of battery pack

(c) The photo of EMS UI

Figure 4.3: The photo of ESS in lab

4.2. Experiment results

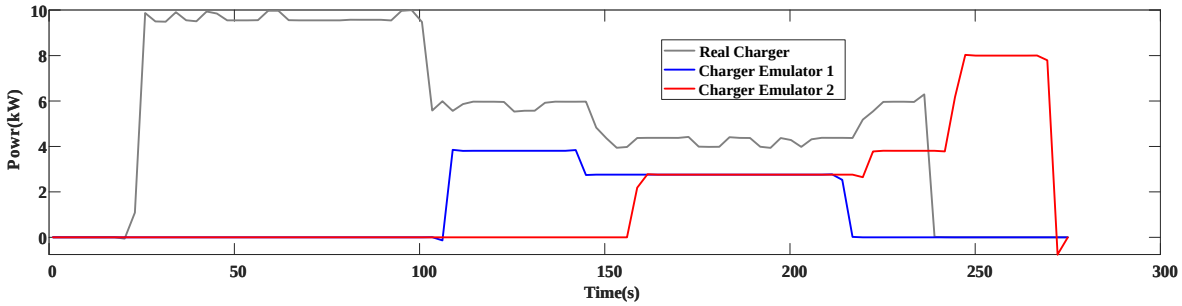


Figure 4.4: The FCR result under practical test

The practical test includes the test of FCR and OLCC method. In FCR test, only three chargers have involved: one DC charger and two charger emulators. For safety concerns, the DC charger will not operates under the maximum power. The output power for FCR test is set to 12kW, and the other two

charger emulators were set to 8kW. The station capacity is set to 10kW. The result is shown in fig 4.4. The result shows the overall power is limited under the station capacity.

As for OLCC test, the output power for DC charger is set to 7kW, the charger emulators are set to 13kW respectivly, the station capacity is set to 30kW. The test time table for different chargers are listed in table 4.5. Then, Fig 4.5 shows the test result of the OLCC method, and the charging demand of three "EVs" are shown in fig 4.6.

Table 4.5: The timetable of charging sequence

Charger Number	Time of Arrival	Time of Departure	Charging Demand
Real Charger	16:55	17:15	1.5 kWh
Charger emulator1	17:05	17:30	3.5 kWh
Charger emulator2	17:00	17:25	1.5 kWh

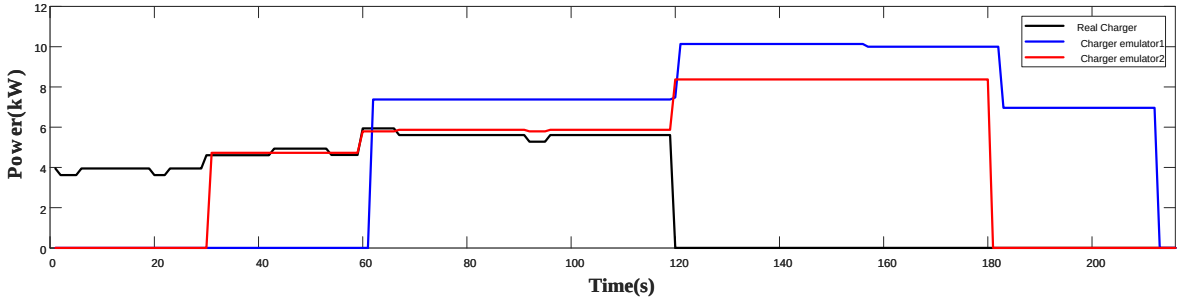


Figure 4.5: The OLCC result under practical test

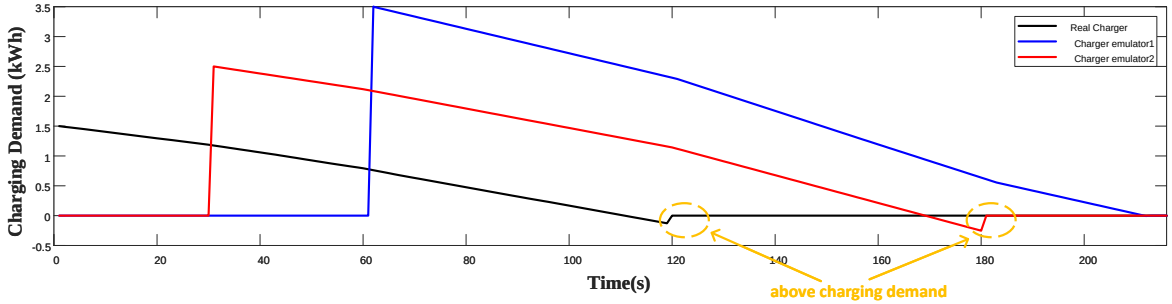


Figure 4.6: The OLCC Charging demand

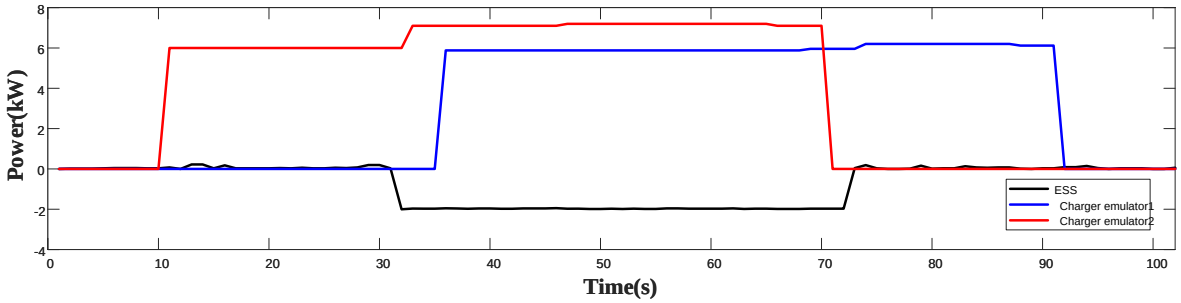


Figure 4.7: The OLCC with ESS result under practical test

At the last the peak shaving method is tested by two charger emulators with ESS. Only the discharging function of peak shaving is tested. The peak shaving starting point is set to 11kW. The peak

shaving mode is automatic. From fig 4.7, it can be seen that if the grid power above the peak shaving starting point, the ESS will be activated to do the peak shaving, and the overall ESS power is limited around 11kW.

4.3. Discussion & evaluation

4.3.1. Performance evaluation of algorithms

According to the results showed in Chapter 3 section 3.5, the performance of the OLCC and FCR can be evaluated. In3.13 and fig 3.10, it can be seen that the OFFCC always be the lowest, because this algorithm is an offline method, the algorithm only calculates once, and all the other parameters such as the timetable of the next day, has to be known before the calculation. Hence, if the EVs' arrival, departure time and the variation of load exactly follows the timetable, the solution must be optimal [20]. This explains why the cost of OFFCC method will be lowest for all simulation. Nevertheless, the timetable is too difficult to be made because the unpredictability of the future. Thereby, an online algorithm OLCC is implemented in this thesis, which can redistribute the charging power of each charger when the system status changes. Furthermore, the speed factor Q induces larger deviation from the optimal solution. Thus, the cost from OLCC method is slightly higher than OFFCC, but OLCC is more feasible to practical scene.

Section 3.5 also illustrated that the FCR method is suitable for low traffic condition compared to OLCC method. The reason is the essence of the OLCC is averaging method, because the charger power for each interval is derived by equation:

$$x_{i_k} = \frac{D_{i_k}}{\Delta T_k} \tag{4.1}$$

Where k represents the k^{th} interval, i^{th} represents the i^{th} EV, D is charging demand and ΔT_k is the length of time at k^{th} interval. So, by referring to this equation, if the interval is small enough and the length of each interval is long enough the final results derived by OLCC method will looks similar to averaging method, this can be proved through fig 3.8, the trend of the OLCC is similar to the AVG method. Moreover, the averaging method cost is higher than FCR method in every simulation. Therefore, as mentioned in section 3.5, the FCR is suitable for charging stations which has a few chargers. On the contrary, the OLCC suitable for large-scale charging stations.

4.3.2. System Cost

In this section, the system cost will be discussed and evaluated. As discussed in section 1.2.1, the data traffic could be the major cost of the system. The cost schematic diagram is displayed in fig 4.8.

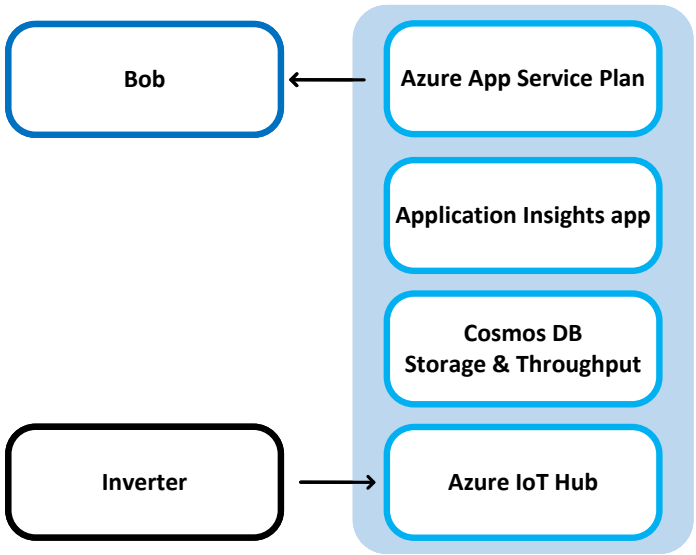


Figure 4.8: The cost from the Microsoft Azure services

There are four main costs from the Azure cloud. Detailed costs are listed in list below:

1. *Azure IoT Hub*: The inverter to cloud service is supported by the Azure IoT Hub service, Microsoft provides multiple tiers with different pricing for different using cases. The detailed tier can be found in [26] and table 4.6. In this thesis, the default standard tier S1 edition is chosen. The pricing details of this plan is: Maximum $400,000$ messages per day for an IoT unit, message meter size up to 4kB. The server is located at western Europe, and therefore the basic cost for this IoT service per month is $21.417€$. Therefore, the cost from Azure IoT Hub is $21.417€$ per month.

Table 4.6: The pricing of Azure IoT Hub

Edition Type	Price per IoT Hub unit (per month)	Total number of messages/day per IoT Hub unit	Message meter size
Free	Free	8,000	0.5KB
S1	$21.417€$	400,000	4KB
S2	$214.161€$	6,000,000	4KB
S3	$2141.603€$	300,000,000	4KB

In addition, to further reduce the system cost, another device can be attached to this plan without generating additional cost. The inverter data update interval is set to 5 seconds in this thesis. The messaged send from this inverter per day would be $60 \div 5 \cdot 60 \cdot 24 = 17280$. Therefore, an additional inverter can be attached into this unit to serve another ESS-Charger system. By doing so the cost can be reduced by half. If more devices is required, the plan can be swap to S2 or even S3, which the cost can be further reduced.

2. *CosmosDB*: As introduced in previous chapters, the Cosmos Db is the database of the *"Bob"* which plays a decisive role in the entire platform, but it comes at a price. The pricing scheme can be divided into two parts in equation 4.2: The first part is the storage cost, and the second part is the data transaction cost.

$$C_{cosmosDb} = C_{storage} + C_{Provisionedthroughput} \tag{4.2}$$

The cost for storage is $0.18€/GB$ per month, the cost for transaction is minimum 400 RU/s (Minimum provisioned throughput) and therefore the cost would be $0.0276€/hour$. Where the RU is the request unit defined by Microsoft, different operations to the database will consume different amount of RU. Hence, the total cost in CosmosDB service will be around $20€/month$.

Since there is only one ESS-charger system have been developed. The operation to the database will not exceed 400RU/s, the cost from the CosmosDB will keeps minimum. However, if more system is put into use, more RU is required for different containers which will leads to more total costs to the system.

3. *Application insights app*: This service will help the system developer to monitor the performance of the whole system, it is necessary to subscribe this service in case there are any unpredictable fault occurs while the whole system is ruining. The pricing is based on how much data is ingested by this app. For this thesis, the insights app pricing method is *Pay as you go* which will cost $2.562€$ per GB, and the data ingested by this App from the *"Bob"* is around 12GB per month. Thus, the cost from this App is about $30.744€$.

4. *Azure App Service Plan*: This service plan will provide a remote virtual machine to support the website or any web API that need to be deployed on cloud. Multiple Apps such as website can be attached to this service plan until the storage of the virtual machine is run out.

Table 4.7: The pricing of Azure service APP

Instance	Core	Ram	Storage	Pay as you go
B1	1	1.75GB	10GB	$0.065€/hour$
B2	2	3.5GB	10GB	$0.129€/hour$
B3	4	7GB	10GB	$0.257€/hour$

In this thesis, the basic plan B1 is selected since the size of the cloud platform is only about 50MB. The pricing scheme is shown in table 4.7. For B1 plan the cost will be around46.8€ per month. Although the cost is relatively higher than the other services, the cost can be further reduced by attaching other Apps to this service until the storage of this virtual machine is full or the CPU percentage reaches upper limit.

In the above analysis, it can be found that the cloud cost is expensive for a single system. The cost fee for a single system is around 120€ However, if multiple systems is attached to the cloud, although the total cost will be increased, the cost for each system can be further reduced compared with a single system on cloud. For instance, if two systems are using the cloud services, the cost for a single system will be reduced by half.

4.3.3. The analysis of communication time delay

Different from the wired signal, the message exchange between the devices and the cloud are wireless. Therefore, the signal transmission speed is lower than the wired transmission. Hence, this section will discuss the influence to the whole system which is caused by the communication delay. Take fig 3.12b as an example, the figure with analysis of the communication delay is drawn in fig 4.9.

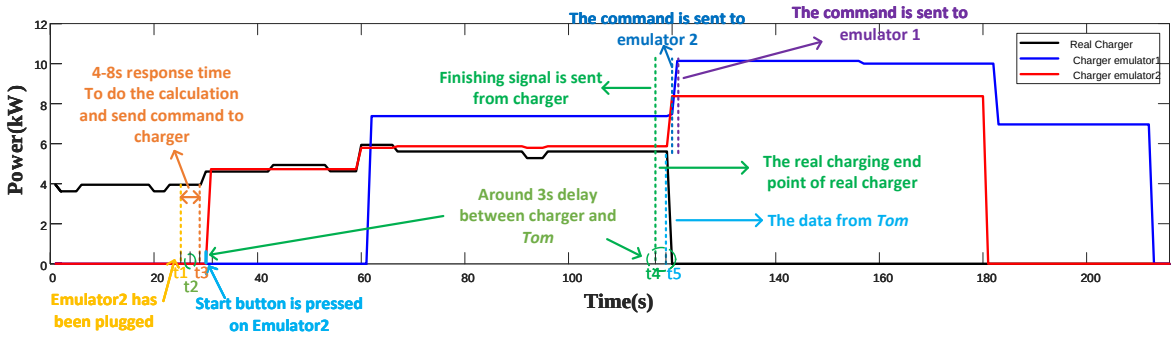


Figure 4.9: The OLCC response analysis

According to fig 4.9, after the charger is plugged into an EV, there is a 4-8 seconds delay until the charger received the power set command from the *"Bob"*, which is t1 to t3 in fig 4.9. Similarly, once an EV has finished the charging process, and emit the *"Finishing"* signal to *"Bob"*, there also will be a small delay until the *"Bob"* gives further instruction to the other chargers. The length of this delay is mainly determined by the complexity of the optimization algorithm (0.5-6 seconds). The low data access speed of the *CosmosDb* also has to be taken into account, the delay is about 50ms. The delay from the calculation can be reduced by using a more powerful server as the base. The delay from the database can be reduced by adding more throughout (RU). However, by doing so, the cost from using the *CosmosDb* will be increased.

Moreover, the API from "*Tom*" only provides the method that change the amount of power for a single charger. Hence, after the calculation, the result for each charger has to be sent one by one, not simultaneously. The results can be seen in colored text *The command is sent to emulator 2* and *The command is sent to emulator 1* in fig 4.9.

In addition, there is an another delay caused by the refresh rate on the charger platform *"Tom"*, this can be seen in green circle in fig 4.9. At time t_4 the charger stops the charging, but the data collected from *"Tom"* remains unchanged until reach the time t_5. The delay time is around 3 seconds. This delay time is depends on the data refresh rate on *"Tom"* and data update rate from charger. Fig 4.10 shows more details about this delay from the charger platform. From the fig 4.10, Gap 1 and Gap 2 shows the message delay from *"Tom"*.

In more details, at time t_1 the user plug the charger into the EV, and the charger will emit this signal to *"Tom"*, the time interval between t_1 and t_2 is around 20ms. Then, the user press the start button and the charger starts charging at t_3, but at the meanwhile, the charger status *"Tom"* such as charging power is still 0, because the data refresh rate on *"Tom"* is 3 second. After 3 seconds at t_4, the *"Tom"* update the charger data. Similarly, at t_5 the EV finished the charging and at t_6 the finishing signal has been emitted. t_9 the data is updated on *"Tom"*.

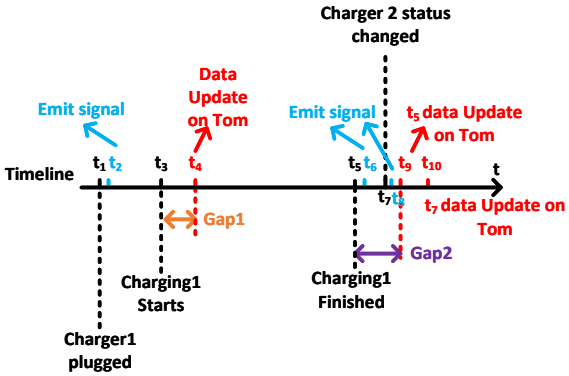


Figure 4.10: The delay from charger platform

This delay will cause miss calculation of the optimization algorithms on *"Bob"*. As introduced in 3.3, once the signal is emitted at t_2 or t_6, the interface *"PowerConfig"* on *"Bob"* will starts the optimization procedure. As shown in fig 3.4 in section 3.3, the first step is to collect the real time data from the other unchanged chargers on *"Tom"*, if the status of the two chargers changes within a short time interval, the wrong real time data will be collected and therefore result in a miss calculation on *"Bob"*. For instance, if 3 chargers are in charging, charger 1 stops charging at t_5, then the charger 2 stops charging at t_7, another signal is emitted to "*Bob*" at t_8.While *"Bob"* processing this second request sent at t_8, the status of charger 1 recorded on *"Tom"* will not be updated until time t_9. So, the fake real time data collected by *"Bob"* at t_8 for charger 1 is still in charging, which is wrong.

Therefore, to let the optimization algorithms on *"Bob"* running correctly, the impact due to this delay has to be reduced. There are two directions to solve this issue:

1. *Set a proper delay on "Bob"*: As for *Gap 1*, the influence caused by *data delay at the beginning* is minimum, since the past condition is the initial condition of the charger (charging starts at 0 power). However, as for the *Gap 2*, to over come this effect, a proper delay has to be set on *"Bob"*. Once the "Finishing" status is received on *"Bob"*, the interface on *"Bob"* will delay for 3 second and then collect the real time data on *"Tom"*.

2. *Increase the data refresh rate on "Tom"*: Increasing the refresh rate on *"Tom"* or update the charger data in a flexible rate: *"Tom"* not only refreshes the data at a fixed rate, but also refreshes the data once the charger status change signal is received from the charger.

4.3.4. Suggestions

By referring to above discussion results, some suggestions can be made.

1. *From algorithm aspect*: This thesis has proved that the algorithm can be put on cloud platform. Therefore, if the performance of server is powerful enough, more advanced algorithms can be implemented on cloud platform to further optimize the charging behavior on EVs.

2. *From the system cost*: Two cloud solutions are introduced in section 1.2, based on the cost analysis, these two solutions are suitable for different scenarios.

   - Scenario 1: Charger-ESS-PV system for home use. This scenario is more suitable for *MES to cloud* method, because the scalability is not necessary under this condition, and the system maintenance fee from cloud for *cloud interconnection* method is high for a single system.
   - Scenario 2: Charger-ESS-PV system for company use. This scenario is more suitable for *cloud interconnection* method, because the company may have multiple charging stations, and therefore the scalability is necessary. Moreover, the maintenance fee from cloud for *cloud interconnection* method can be reduced by supporting multiple cloud platforms.

3. *From the time delay*: The signal processing order has to be considered carefully when designing such a cloud based charging management system.

5

Conclusion & Future work

5.1. Conclusion

In this thesis, a cloud-based charging management platform is proposed. This platform is able to monitor the status of the chargers and control the charging behavior of different chargers in the charging station. The ESS and its control also have been integrated into this system.

The development of the cloud platform is divided into 3 parts: Website *"Bob"* design , algorithm design and experimental validation. The conclusions can be made:

1. From the content introduced in Chapter 2, the design of *"Bob"* can be emphasized as follows:
   - The charger monitoring, charger grouping and charger power controlling functions are implemented by using the interfaces provided by *"Tom"*.
   - The MVC framework is selected to be the basic framework of the *"Bob"*. The database is set to record the past data from the chargers and the charging algorithm settings for different charger groups.
   - The IoT of ESS devices have been done based on Microsoft Azure services.
2. From the algorithm design illustrated in the chapter 3, a quick review can be draw:
   - Two charging algorithms *"FCR"*, *OLCC* have been developed in this chapter, and the simulation results of these two algorithms are given at the end of the chapter 3.
   - A simple ESS management peak shaving method was implemented.
   - The theory of how algorithm computation is running on cloud was explained by introducing the interface *"PowerConfig"* and its operation principle.
3. From experimental results showed in Chapter 4, it can be seen that:
   - The cloud is able to manage the charger behavior remotely with an acceptable delay. The developed charging algorithms and peak shaving method functions well during the test.
   - The cost generated from the cloud side has to be considered carefully. The cost can be reduced by developing more cloud platforms to more customer. The cost also can be reduced by reducing the data transmission rate or the size of the data. Nevertheless, the lower transmission rate may longer the system delay.
   - The time delay of the cloud communication may bring some issues to the system, such as miss calculation. However, this influence can be minimized by improving the data transmission rate, and using a better server. However, the system cost will be increased.
   - The trade-off between the cost-effective and high performance has to be considered carefully based on different application conditions.

5.2. Future work

Since this system is still in early stages, there are many aspects that can be improved in future, which are:

- *Develop more advanced charging algorithms based on this platform*: In this thesis, only two charging algorithms have been developed, and both of them are low-level algorithms. Currently, many up-to-date algorithms which combines the future forecasting or deep learning are proposed by other papers such as [27, 28].
- *Add photovoltaic into the system*: Due to the consideration of the workload and the time, this thesis only discussed the condition that the chargers only combined with ESS. However, in practical the customers wants to buy ESS-PV system together. Hence, the PV system and its control method need to be developed in future. The final view of the whole system is shown in fig 5.1

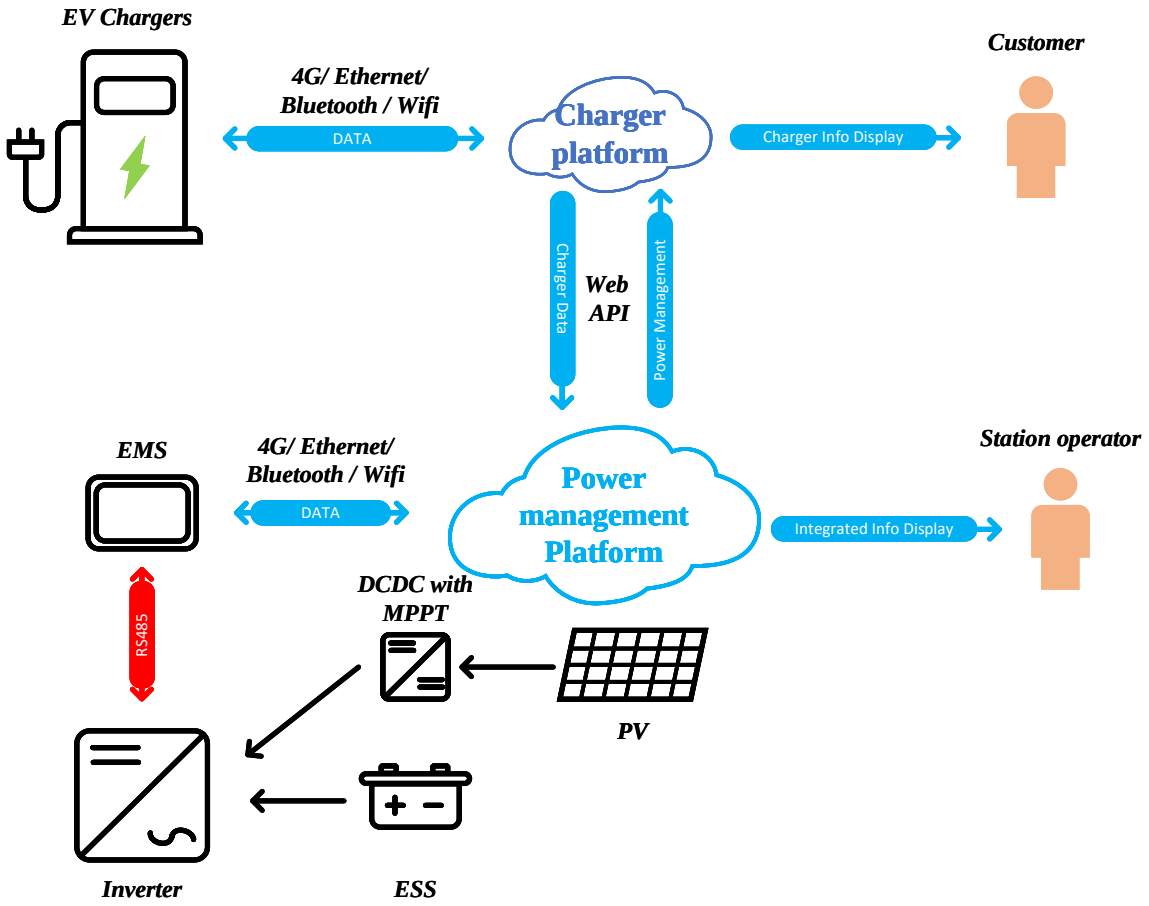


Figure 5.1: The preview of the complete system

- *Refine the UI design on "Bob"*: The UI on "*Bob*" can be further developed, such as adding more figures to make the data look more intuitive. In addition, the robustness of the platform also has to be reinforced, for instance, by adding role management system to the platform.
- After well-developed and tested of this cloud platform, it can be used for commercial purpose.

A

EMS design

In the body of the thesis, only the JAVA program of the industrial computer for inverter was introduced. therefore, this chapter will show how the message is collected from the inverter and been sent to JAVA through the Qt program.

As mentioned in section 2.2.3, Modbus RTU protocol is applied to exchange the data between EMS (industrial computer) and the inverter. A complete Modbus message will contains following information in table A.1, the definition of each register is defined by the manufacture of the inverter.

Table A.1: Modbus RTU format

$Name$	$Length(bits)$	$Function$
Address	8	Slave address
Function	8	Function code
Data	n× 8	Data + length will be filled depending on the message type
CRC	16	Cyclic redundancy check

For example, if the EMS want to read the battery voltage, the EMS will send command ”01 03 31 40 00 01 8B 22”. Where 01 is the address of the inverter, 03 is the function code that represent this action is a read action, 31 and 40 respectively represent the high and low bits of the register address, 00 and 01 respectively represent the high and low bits of the amount of the register that going to be read, 8B and 22 are the CRC low and high bits respectively.

The flow chart of the Qt program is shown in fig A.1. This program will run multiple threads together to ensure that the collected data can be displayed on the screen and sent to java in a short interval. Finally, the designed UI for the industrial computer is shown in fig A.2. The user is able to view the inverter data through the screen of the industrial computer and the settings of the inverter can be modified through the screen as well.

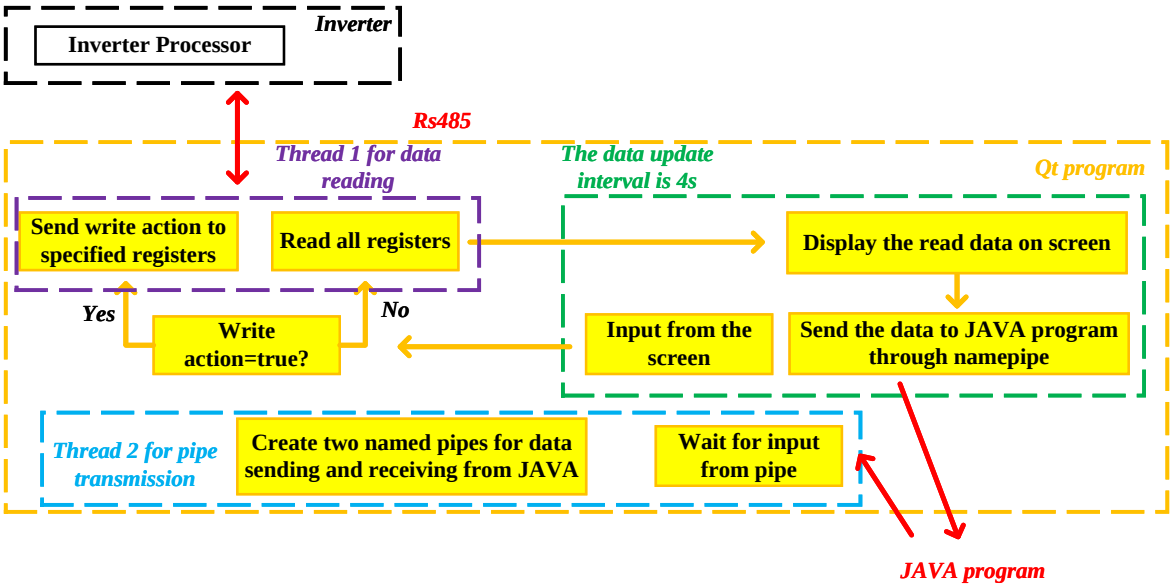


Figure A.1: The flow chart of Qt program

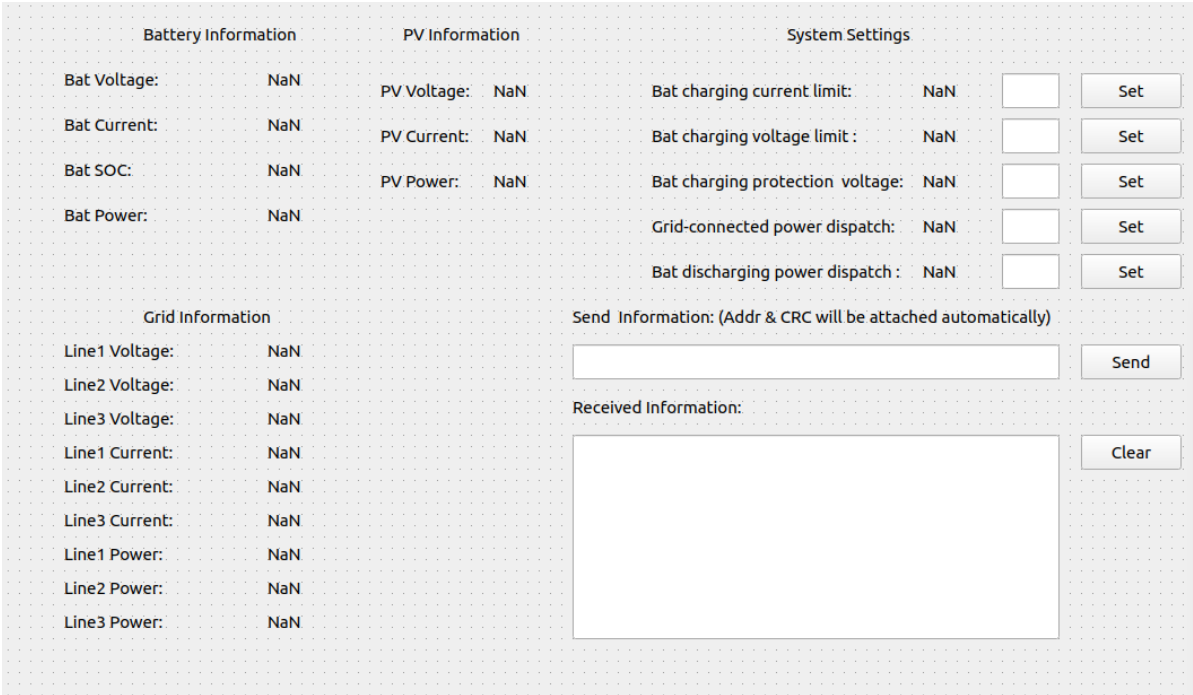


Figure A.2: The UI of industrial computer

B

Part of source codes of Bob

```
module.exports = function (context, IoTHubMessages) {
     var id = "";
    var invId = "";
    var batV = "";
    var batC = "";
    var line1V = "";
    var line2V = "";
    var line3V = "";
    var line1A = "";
    var line2A = "";
    var line3A = "";
    var timeStamp = "";
    var workMode = "";
    var batChargingC= "";
    var invOutP = "";
    var batOutP = "";
    var batType = "";
    var batChargingV = "";
    var batOVP= "";

    IoTHubMessages.forEach(message => {

    if(message.InverterId !== "" \&\& message.Id !== "")
    {
        var output = {
            "id": message.Id,
            "invId": message.InverterId,
            "batV": message.BatteryVoltage,
            "batC": message.BatteryCurrent,
            "line1V": message.Line1V,
            "line2V": message.Line2V,
            "line3V": message.Line3V,
            "line1A": message.Line1A,
            "line2A": message.Line2A,
            "line3A": message.Line3A,
            "timeStamp": message.TimeStamp,
            "workMode": message.WorkMode,
            "batChargingC": message.BatteryChargingCurrent,
            "invOutP": message.InverterOutputPower,
            "batOutP": message.BatteryOutputPower,
            "batType": message.BatteryType,
            "batChargingV": message.BatteryChargingVoltage,
            "batOVP": message.BatteryOvervoltageProtection
        }
        context.bindings.testMeterDB = output;
        var date = Date.now();
        var tableMsg = {
             "partitionKey": message.InverterId,
             "rowKey": date+'',
             "MsgCount": message.length,
             "data": JSON.stringify(message)
        }
        context.bindings.outputBlob = tableMsg;
    }
    });


    context.done();
};
```

Listing 7: The srouce code of Azure function for *"Bob"*

A standard message format that the *"Tom"* POST to interface *"PowerConfig"* on *"Bob"* should be:

```
{
    "ChargerId":"QixiangTest001",
 "SystemStatus": "2021/09/21 17:00:30",
  "ChargingDemand":"3.5",
  //ChargingDemand by Qixiang
  "LeavingTime":"2021/09/21 17:30:30",
  //LeavingTime by Qixiang
    "TimeStamp": "2021/09/21 17:05:30",
 "Plugs":[
 {
            "PlugId": "PlugA",
            "Status": "Preparing",
            "ErrorCode":"No Error",
            "MaxActivePower":"13kW",
            "MeterValue":[
                        { "value":"4.9",
                        "format":"raw",
                        "measurand":"ActivePower",
                        "unit":"kW"},
                        { "value":"0",
                        "format":"raw",
                        "measurand":"ReactivePower",
                        "unit":"kVar"},
                        {"value":"452",
                         "format":"raw",
                         "measurand":"ActiveConsumptionEnergy",
                         "unit":"kWh"},
                        {"value":"360",
                        "format":"raw",
                        "measurand":"DCVoltage",
                        "unit":"V"},
                        {"value":"13.61",
                        "format":"raw",
                        "measurand":"DCCurrent",
                        "unit":"A"},
                        {"value":"60",
                        "format":"raw",
                        "measurand":"SoC",
                        "unit":""}
                        ]
    }

}
```

Listing 8: The srouce code of Azure function for *"Bob"*

C

Simulation data

The simulation results for heavy traffic condition can be found in fig C.1 and fig C.2

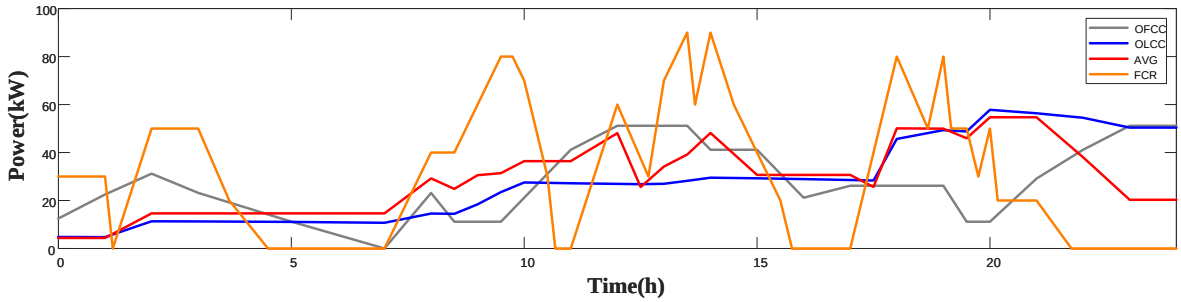


Figure C.1: The total charging power with different methods for a day under heavy traffic

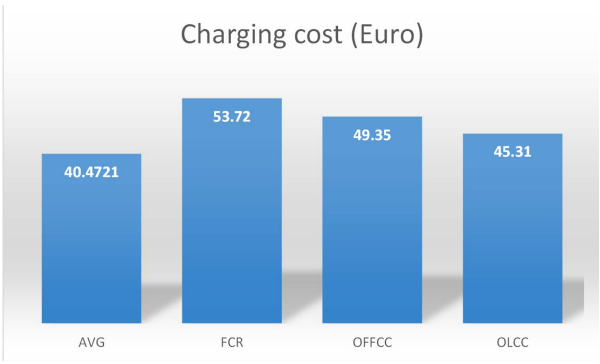


Figure C.2: The total cost of different methods for heavy-term simulation

Bibliography

[1] *Global electric passenger car stock*, IEA, Paris (2010).

[2] S. Deb, K. Tammi, K. Kalita, and P. Mahanta, *Impact of electric vehicle charging station load on distribution network,* Energies **11**, 178 (2018).

[3] L. Wang, Z. Qin, T. Slangen, P. Bauer, and T. Van Wijk, *Grid impact of electric vehicle fast charging stations: Trends, standards, issues and mitigation measures-an overview,* IEEE Open Journal of Power Electronics (2021).

[4] S. M. Alshareef and W. G. Morsi, *Impact of fast charging stations on the voltage flicker in the electric power distribution systems,* in *2017 IEEE Electrical Power and Energy Conference (EPEC)* (IEEE, 2017) pp. 1–6.

[5] C. Jiang, R. Torquato, D. Salles, and W. Xu, *Method to assess the power-quality impact of plug-in electric vehicles,* IEEE Transactions on Power Delivery **29**, 958 (2013).

[6] A. Ul-Haq, C. Cecati, K. Strunz, and E. Abbasi, *Impact of electric vehicle charging on voltage unbalance in an urban distribution network,* Intelligent Industrial Systems **1**, 51 (2015).

[7] K. Clement-Nyns, E. Haesen, and J. Driesen, *The impact of charging plug-in hybrid electric vehicles on a residential distribution grid,* IEEE Transactions on power systems **25**, 371 (2009).

[8] A. Amin, W. U. K. Tareen, M. Usman, H. Ali, I. Bari, B. Horan, S. Mekhilef, M. Asif, S. Ahmed, and A. Mahmood, *A review of optimal charging strategy for electric vehicles under dynamic pricing schemes in the distribution charging network,* Sustainability **12**, 10160 (2020).

[9] K. Chaudhari, A. Ukil, K. N. Kumar, U. Manandhar, and S. K. Kollimalla, *Hybrid optimization for economic deployment of ess in pv-integrated ev charging stations,* IEEE Transactions on Industrial Informatics **14**, 106 (2017).

[10] T. S. Bryden, G. Hilton, B. Dimitrov, C. P. de León, and A. Cruden, *Rating a stationary energy storage system within a fast electric vehicle charging station considering user waiting times,* IEEE Transactions on Transportation Electrification **5**, 879 (2019).

[11] S. Negarestani, M. Fotuhi-Firuzabad, M. Rastegar, and A. Rajabi-Ghahnavieh, *Optimal sizing of storage system in a fast charging station for plug-in hybrid electric vehicles,* IEEE transactions on transportation electrification **2**, 443 (2016).

[12] *What is azure iot hub,* (), accessed: 16-08-2021.

[13] *Third place energy,* Accessed: 16-08-2021.

[14] *Azure iot samples for c sharp (.net),* Accessed: 20-08-2021.

[15] M. Jailia, A. Kumar, M. Agarwal, and I. Sinha, *Behavior of mvc (model view controller) based web application developed in php and. net framework,* in *2016 International Conference on ICT in Business Industry & Government (ICTBIG)* (IEEE, 2016) pp. 1–5.

[16] L. To and F. T. Reenskaug, *Thing-model-view-editor an example from a planningsystem,* (1979).

[17] *Iothubeventcallback interface,* Accessed: 25-08-2021.

[18] *Interface runnable,* Accessed: 25-08-2021.

[19] *iot-using-cosmos-db,* (), accessed: 31-08-2021.

[20] W. Tang, S. Bi, and Y. J. Zhang, *Online coordinated charging decision algorithm for electric vehicles without future information,* IEEE Transactions on Smart Grid **5**, 2810 (2014).

[21] Z. Ma, D. S. Callaway, and I. A. Hiskens, *Decentralized charging control of large populations of plug-in electric vehicles,* IEEE Transactions on control systems technology **21**, 67 (2011).

[22] Y. He, B. Venkatesh, and L. Guan, *Optimal scheduling for charging and discharging of electric vehicles,* IEEE transactions on smart grid **3**, 1095 (2012).

[23] M. G. Flammini, G. Prettico, A. Julea, G. Fulli, A. Mazza, and G. Chicco, *Statistical characterisation of the real transaction data gathered from electric vehicle charging stations,* Electric Power Systems Research **166**, 136 (2019).

[24] S. Chen and L. Tong, *iems for large scale charging of electric vehicles: Architecture and optimal online scheduling,* in *2012 IEEE Third International Conference on Smart Grid Communications (SmartGridComm)* (IEEE, 2012) pp. 629–634.

[25] *Open energy managment system,* Accessed: 31-09-2021.

[26] *Azure iot hub pricing,* Accessed: 05-10-2021.

[27] F. Hafiz, M. Awal, A. R. de Queiroz, and I. Husain, *Real-time stochastic optimization of energy storage management using deep learning-based forecasts for residential pv applications,* IEEE Transactions on Industry Applications **56**, 2216 (2020).

[28] G. F. Savari, V. Krishnasamy, J. Sathik, Z. M. Ali, and S. H. A. Aleem, *Internet of things based real-time electric vehicle load forecasting and charging station recommendation,* ISA transactions **97**, 431 (2020).