POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Comfort parameters auto-configuration system for the automotive domain

Proof of concept realization

Supervisor
prof. Massimo Violante

Candidate
Antonio Ciardo
matricola: 240816

Internship Tutor
dr. Silvia Cataldo

Academic year 2017-2018

Summary

Today, the automotive domain is changing fast, cars are becoming always more connected, making possible the realization of new features to be proposed to the final user. In this thesis work, I studied the integration of a car system able to configure automatically some car comfort parameters based on the user preferences.

Acknowledgements

I would like to express my sincere gratitude to my internship tutor dr. Silvia Cataldo and the whole Reply company for giving me the possibility to realize this thesis work.

Contents

List of Figures

Chapter 1

Introduction

1.1 The case study

Nowadays, the automotive domain is changing fast. The always lower hardware cost has made possible the integration of many embedded systems in a vehicle, opening the way to the internet connectivity. Connected cars are becoming a strong reality, and it is very likely that, by 2025, all cars will have an internet connection, as shown in the image 1.1.

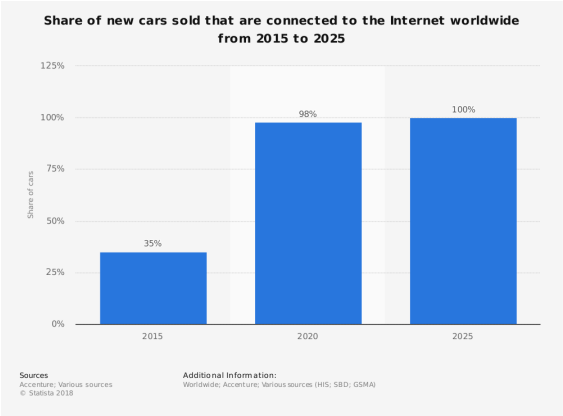

Figure 1.1. Connected cars forecast [1]

There are many reasons that justify the embedding of network capabilities in cars, including the fact that it gives the opportunity to:

- **enhance the user experience:** connected cars can offer all the features to entertain and assist users. It enables things like car remote control, built-in navigation system with traffic updates, voice assistant, etc.

- **release software "over the air" (OTA) update:** in case of software bugs, car manufacturers can update remotely the software presents in the many in-vehicle embedded systems, so users are not forced to bring car to repair. The same principle can be used also to upgrade a vehicle with new features.
- **collect diagnostic and maintenance data:** car manufacturers can analyze the collected data to provide a better user experience or to detect some flaws.

Therefore, the general trend in automotive domain is to invest highly into complex electronic devices (embedded systems) and connections with the outside world, even if there are some important security drawbacks to be addressed.

1.2 State of the art

Modern vehicles have embedded a large quantity of electronic devices used to control almost everything in a car. Such devices are called electronic control units (ECU), and they are basically embedded systems, i.e., computers designed to run specific programs. There are 70 ECUs on average in a new car [2], that are typically grouped in different functional domains [3]:

- **power train domain**: it includes systems related to the vehicle propulsion (engine, transmission, etc.)
- **chassis domain**: it includes systems related to the four wheels (steering and braking)
- **body domain**: it includes systems that do not belong to the vehicle dynamic (seat control, door lock, lighting, trunk release, rain sensor, air conditioning, etc.)
- **infotainment/HMI domain**: it includes systems related to the communication between car and user (display, etc.)

These domains communicate with each others and with the outside world through a central gateway, i.e., a system that performs the translation among the different communication protocols present in the in-vehicle network. Its role is not limited only to interface the different protocols, but it also includes activities like the message routing and filtering on the network, the

management of OTA updates, etc.
In the image 1.2 is represented a simplified block diagram of a typical in-vehicle network.

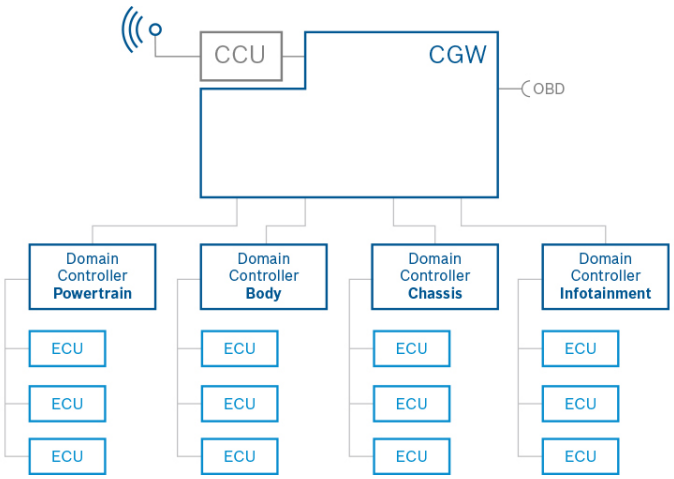

Figure 1.2. Gateway CGW [4]

In-vehicle communication protocols ECUs inside car communicate with each other and with sensors/actuators using different protocols, each one targeting a different constraint. These protocols are:

1. **CAN:** it is a serial, multiple master protocol with bit rate up to 1 Mbit/s. It is able to manage efficiently real-time control messages with a low routing cost, since it uses only two wires for its differential signal. It is used for medium speed applications, mainly inter ECUs communications.

2. **LIN:** it is a serial, single master protocol with bit rate up to 20 kbit/s. It is a low cost communication protocol (only one wire) based on the UART byte interface. It is used in sub-network where the efficiency and performance of CAN is not required, like sensors and actuators.

3. **FlexRay:** it is a serial protocol with bit rate up to 10 Mbit/s. It can have one or two channel (1 channel is composed by a twisted pair of wires) and it is used for real-time and safety critical applications.

4. **Ethernet:** it is a custom implementation of the Ethernet protocol tailored for the automotive domain. It has a bit-rate up to 100 Mbps and it enables high bandwidth communications.

For what concern the communication with the external world, the trend is to provide in-vehicle Wi-Fi, 3G, 4G, LTE. [6]

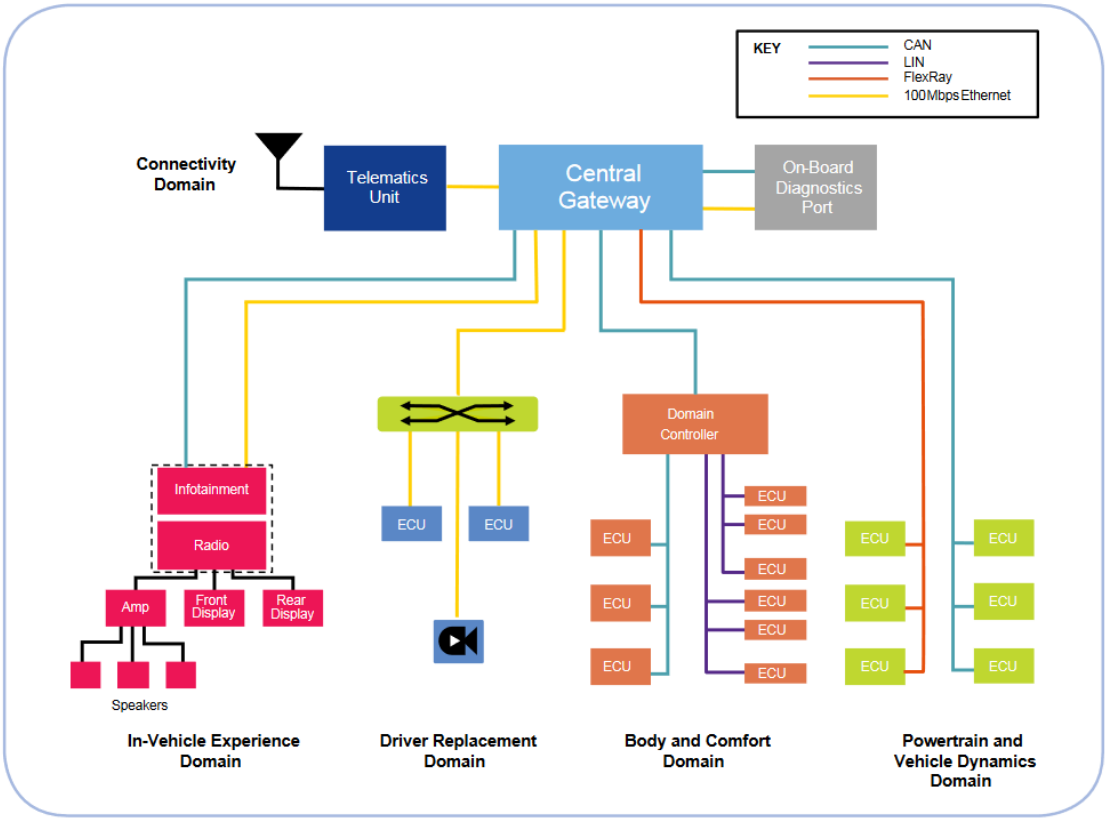

Figure 1.3. In-vehicle network [7]

1.3 Thesis goal and methods

The goal of this thesis is to realize a proof of concept of a system that is able to recognize the user that wants to enter the car, unlock it and successively set automatically some car comfort parameters based on the user preferences. The ideal scenario is the luxury car domain. In order to implement it, I organized my work in different steps.

Step 1 During the first step, I analyzed the state of the art about the in-vehicle network and the in-car internet connectivity in order to both understand how the same problem is addressed in the real world, and to see

if there were already some implementations. I found out that luxury cars already have similar systems to the one I would have implemented. They typically use a smartphone app to recognize the user, who can control remotely the car always through the app. However, they do not implement the automatic set of the comfort parameters based on the particular user.

Step 2 I studied the problem from an architectural point of view, trying to stay as abstract and independent as possible from the eventual hardware I would have used. During this stage, I elaborated the architecture shown in figure 1.4. There are four key components: an ECU, a server, a smarthphone app and a module able to interface the ECU with the server.

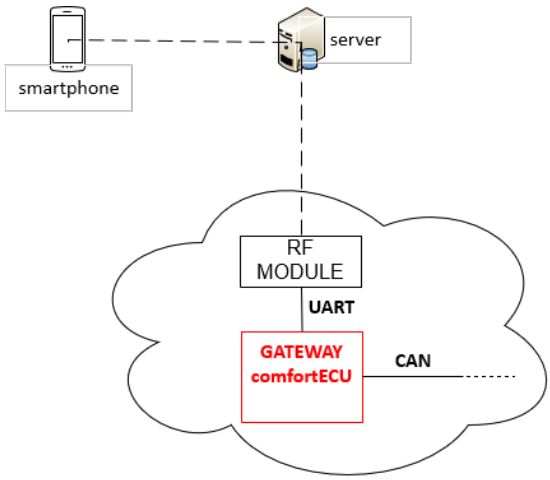

Figure 1.4. Architectural model

Step 3 I decided which kind of hardware I would used for my implementation.

Step 4 After having decided the architecture and the hardware of my system, I started the development phase, writing the software running inside the ECU, the Server and the app.

Step 5 During this last step, I perform all the test necessary to ensure the correct behaviour of my system.

1.4 Chapters overview

The content of the following chapters is described briefly below:

- Chapter 2: it addresses the general implementation of the system
- Chapter 3: it is focused on the Server implementation aspects
- Chapter 4: it is focused on the ECU implementation aspects
- Chapter 5: it is focused on the Application implementation aspects
- Chapter 6: it contains the conclusions

Chapter 2

The system

2.1 Requirements

The designed system is composed by three key components all necessary for the correct communication between the user and the car. These components are:

1. **ComfortECU:** it receives commands from the user and performs the requested action.

2. **Application:** the car user sends commands and receives information about the ECU thanks to a smartphone application.

3. **Server:** it resides between ECU and the application, managing the communication between them.

All three work coordinately in a determinate set of scenarios which are:

- **Toggle scenario:** The user clicks the unlock/lock button on the app, triggering the unlocking/locking of the car along with the application of the comfort parameters in case of unlocking.

- **Save scenario:** The user clicks the save button on the app in order to save the current comfort parameters configuration on the server.

- **Apply scenario:** The user clicks the apply scenario button on the app, requesting to the ECU to download from server the current comfort parameters and apply them.

- **Tracking scenario:** The user can track the position and see information about the lock/unlock status of his car, since the ComfortECU saves periodically on server its GPS coordinates along with the car status.

For this purpose, there is the need to manage a set of users, therefore the system has the concept of Admin user, that is in charge to:

- Insert a new user account into the database
- Insert a new car in the database
- Link user to his car
- Delete a user account
- Create a new Admin account

While, the normal user has the possibility to create his account and associate more profiles to it, allowing to have more than one set of comfort parameters saved on database that could be applied.

User account creation The user account creation can be done, in principle, in two different ways: either the car vendor gives to the user a unique ID identifying his car, so the user during the account creation can insert this id and link his account to his car, or the binding between user account and his car can be delegated to an Admin user. I choose the second way, because it should be easier to apply from the user point of view, since he does not have to insert manually any ID.

2.2 Implementation choices

2.2.1 ComfortECU

The micro-controller (MCU) emulating the ComfortECU must support the CAN protocol in order to send messages to other ECUs inside the car. Therefore, the search was limited to all MCUs with the CAN controller. The three considered boards were:

- *STM32F407* discovery board by *STMicroelectronics*
- *S32K144* evaluation board by *NXP Semiconductors*

- *LPC11C24* by *NXP Semiconductors*

In the end, I chose the *S32K144* principally for three reasons. First, it has the CAN transceiver already on-board, which is lacking in the case of the *STM32F407*. The second reason is that *LPC11C24* has only 8 kbyte of RAM memory contrary to the 64 Kbyte of the *S32K144*. Last, the *S32K144* belongs to the new class of microcontrollers designed by *NXP* for the automotive domain. In the table 2.1 are condensed the main characteristics of the three board.

	STM32F407	**S32K144EVB**	**LPC11C24**
CORE	ARM Cortex M4 (fmax = 168MHz)	ARM Cortex M4F (fmax = 112MHz)	ARM Cortex M0 (fmax = 50MHz)
FLASH	1 Mbyte	512 kbyte	32 kbyte
RAM	192kbyte	64 kbyte	8 kbyte
CAN	2 CAN controller No CAN transceiver	3 CAN controller UJA1168 CAN transceiver	1 CAN controller TJF1051 CAN transceiver

Table 2.1. Board comparison

Regarding the internet connectivity, the board must be attached to a cellular module to establish the ECU-Server communication, so I used the low cost *SIM808* GPRS/GSM/GPS module designed by *SIMCom*. Board interacts with *SIM808* sending AT commands via UART protocol, which are standardized instructions used to control a modem. Between *S32K144* and other emulated car ECU the communication protocol used is CAN. In summary, the comfortECU communicates with:

- the server by means of a GPRS module, controlled with AT commands
- the other emulated ECUs using CAN protocol

Ideally, it should work taking comfort messages from the server and routing them to to the correct emulated ECU. Identically, it should receive CAN messages from the other ECUs and send them to the database. Another kind of message that the ComfortECU should manage is the lock/unlock command so it is performing a role similar to the central gateway described in section 1.2.

2.2.2 Server

The backend is implemented using Firebase powered by Google, that is actually a server-less solution. Server-less means that the cloud provider (Google

in this case) offers to the developer a series of cloud services already implemented, while keeping hidden the logic managing the server on which these services run. Thus, the term is not indicating the development of a system that does not use an actual server, but that the developer doesn't need to worry about the configuration and maintenance of the server and he can use directly the provided cloud functionalities. Firebase can be classified as both BaaS and FaaS system.

- **BaaS:** it stays for Backend-as-a-Service, a term that refers to a system providing a set of cloud functionalities, already implemented, that are usually needed in a backend environment. The developer can integrate these functionalities in his application thanks to pre-built SDKs. Among all the features provided by Firebase, in this work I use:
  - The Firebase Real-Time database: NoSQL database stored in Google Cloud. The underlying logic of the database creation and management is hidden to the developer that has only to use the database through provided SDK or using the REST API.
  - Firebase Authentication service that implements the basic logic for the management of user registration and authentication.
- **FaaS:** it stays for Function-as-a-Service, a term that refers to a system providing to the developer a cloud space where hosting custom back-end code (functions), i.e., code written by the developer. The difference with normal back-end code is that in FaaS, the custom code must be structured in single stateless functions, event-triggered that are managed by the provider (the provider decides how many resources must be allocated for a determinate function). Firebase calls this kind of functions "Firebase Cloud Functions" and they can interact with the Firebase real-time database and the Firebase Auth service through the provided SDKs.

Thus, the system is able to authenticate the user and run back-end code to modify the database. Moreover, Firebase communicates only with HTTP connection encrypted with SSL (HTTPS). It seems a good choice for developing a proof of concept, since it allows to build a complete functioning system in less time.

2.2.3 User and Admin Application

The user application is an Android application needed to make possible the interaction between user and server, while the Admin app is a simple

command-line interface (CLI), written in Javascript, with a set of prebuilt functions to manipulate the database and manage the users.

2.3 Architecture

In figure 2.1, the architecture of the designed system is represented.

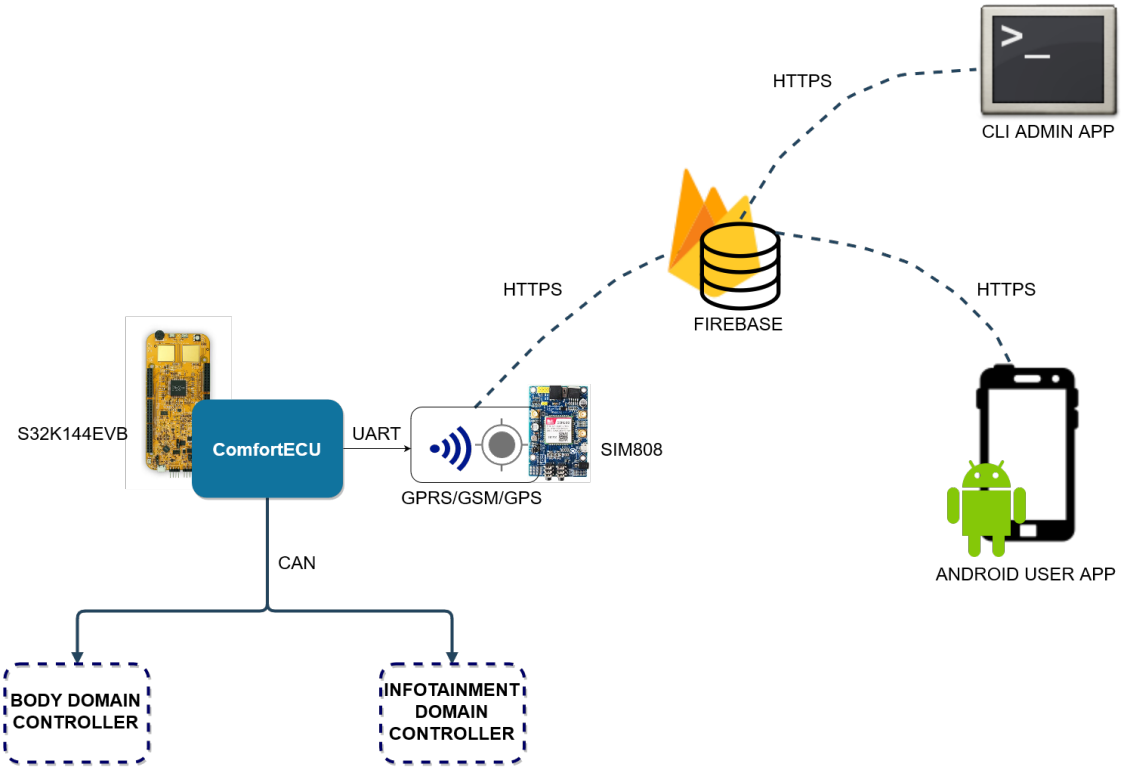

Figure 2.1. System architecture

Chapter 3

Server

3.1 Firebase services

The server is used for connecting the user mobile application with the ComfortECU of his car. It is a server coupled with a database providing a set of functionalities that can be used by:

- The ComfortECU
- The user with his Android application
- The admin user with his command line interface

As I explained in section 2.2.2, I decided to use Firebase for the implementation of the back-end, and the services integrated in my system are:

- Firebase Real-Time Database for the database
- Firebase Auth for the user management
- Firebase Cloud Function to deploy custom back-end code

Firebase Real-Time Database Firebase real-time database is a NoSQL database hosted on the Google cloud and structured as a JSON tree, where each data stored in the JSON is threated as JSON object. Such object is a key-value couple, where the key must be a string, while the value can be string, number, another object (nested), array, boolean or null [8]. The code below gives a visual example of how this kind of database can be structured:

```
{
    "users": {
        "user1" : {
            "firstname" : "Eric",
            "lastname" : "Cartman"
        },
        "user2" : {
            "firstname" : "Walter",
            "lastname" : "White"
        }
    }
}
```

Firebase Auth Firebase Auth is a service provided by Firebase and it can be used for the management of the users from their registration to the deletion of their account. A project that use Firebase as back-end and that offers the user registration through the Firebase Auth service, will have the list of registered user stored in the Firebase cloud as 'Firebase User' objects. This object has at most four attributes, that are first populated during the user registration (then the user can updates them). These attributese are a unique ID identifying the user, an email address, a name and a photo URL. Therefore, Firebase Auth manages the user registration and give to the developer the opportunity to implement in his app different registration methods like the registration with email and password, with external provider (e.g., Google SignIn) and last by defining custom authentication system. Once a user is registered, the service keeps track of the state of the user authentication and reacts opportunely to sign-in, sign-out and other kind of events.

Firebase Cloud Functions Firebase lets the developer writes his custom code and deploys it on Firebase cloud, allowing to run this back-end code when triggered by specific events. Therefore, the point is that the developer can write a set of functions event-triggered and upload them on the Firebase Cloud. There can be used different kind of events to trigger the functions, including:

- **HTTP triggers:** the functions will be associated to an HTTP endpoint (an URL) and they can be invoked directly by sending an HTTP request to their URL.

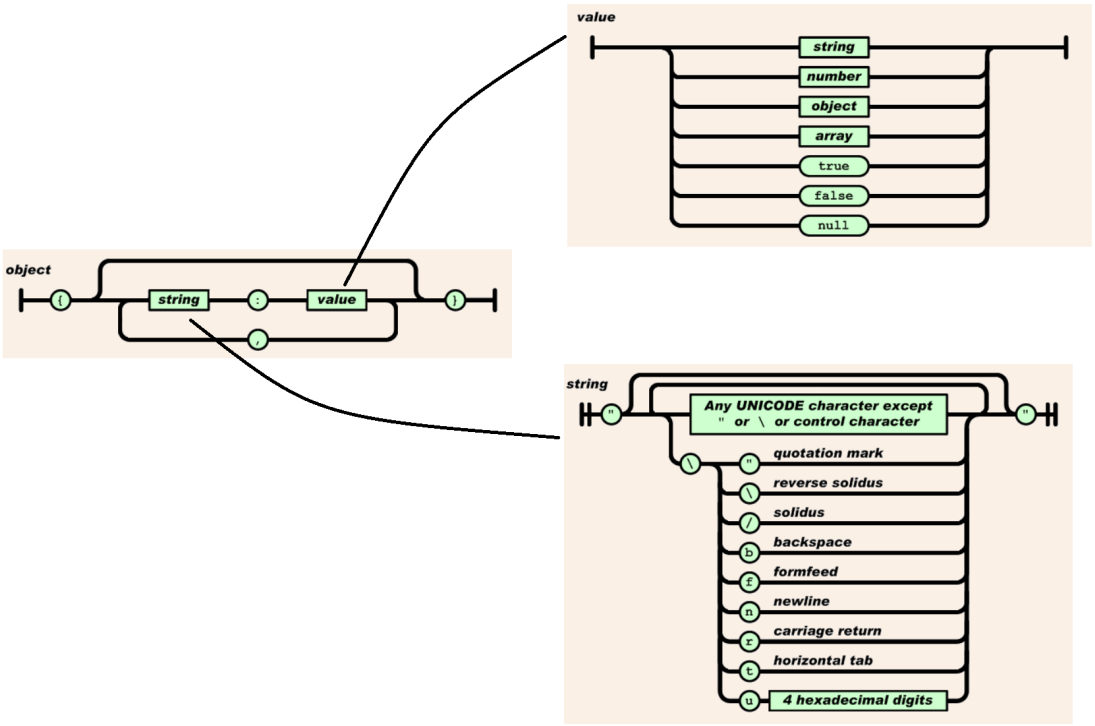

Figure 3.1. JSON object visual representation [12]

- **Realtime Database Triggers:** a function can be triggered by a particular event happening in the Firebase database. For example, a function can listen for write events on any database location, so when on that location something is created, updated or deleted, the function is triggered.
- **Firebase Authentication Triggers:** same as Realtime database triggers, but this time, the events that can trigger the cloud functions are related to the Authentication service. For example, a function listening for creation of new user will be triggered when a new user is created.

3.1.1 Interaction with Firebase services

Projects that have Firebase as backend must have some ways to communicate with the services provided. Most times, there are 3 methods to interact with them:

1. Through a REST API

2. Through a Platform-targeted Firebase SDK
3. Through a Platform-targeted Firebase Admin SDK

I will give an overview about each method that the developer can use for the interaction with each service.

REST API It is a set of functions that follows the REST paradigm. At this point, I must introduce what is REST. The term stays for 'REpresentational State Transfer' and it is an architectural style for networked computers that introduce a set of rules and constraints in how the interaction among these computers should happen. The main constraints are:[9] [10]

1. server and client code must be kept separated
2. the server must not store the state of the system (stateless). All the server need to know to carry out successfully the client request must be stored in the request itself
3. server and client must agree on a well defined format for the exchanged resource. Typically the chosen formats are JSON and XML.

Since REST is defined for networked computer, these functions must be accessible on the network, for example by exposing them as HTTP endpoints[1] callable by anyone that knows these endpoints (endpoints means URLs).

In the case of Firebase database, its REST API makes each location of the database accessible as an HTTP endpoint, appending to this URL the word ".json" [11]. Supposing that the root of the database endpoint is:

```
https://databaseroot.com
```

and supposing that the database structure is the following one:

```
{
  "users" : {
    "user1": {
        "firstname" : "Joss",
```

[1]It must be specified that REST doesn't imply HTTP to work. It could be implemented on any network protocol, but Firebase uses HTTP, therefore only REST over HTTP is considered here.

```
        "lastname" : "Brewer"
    }
  }
}
```

in order to retrieve user1 from the JSON tree I must perform an HTTPS request (plain HTTP is refused by firebase) with the following parameters:

- URL requested: `https://databaseroot.com/user/user1.json`
- HTTP method: `GET`
- Accept-header: `application/json`

It will return status response 200 and the json:

```
{"firstname":"Joss", "lastname":"Brewer"}
```

Instead, in order to update (modify) it, by adding the age, the request could be:

- URL requested: `https://databaseroot.com/user/user1.json`
- HTTP method: `PATCH`
- Content-type: `application/json`
- Body data: `{ "age" : 36 }`
- Accept-header: `application/json`

It will return status response 200 and the json: `{"age":36}`.

There exists a REST API also for the Authentication service, that allows things like user creation, sign in, etc., in a RESTful manner.

Last, for Firebase Cloud functions the discussion is different, since they are only custom back-end code uploaded by the user. They can be exposed as HTTP endpoint, but this does not mean that the functions follow the REST paradigm: it is the developer that could use these functions to built a REST-compliant API. In conclusion, Cloud functions can be invoked through HTTP request, but the invocation could not be RESTful.

The REST API is usefull when other methods provided for access the Database

or the authentication service can't be applied. For example, the ComfortECU can't use the SDK but it could use the REST API to access directly the Firebase Database by simply sending HTTPS request in the correct way. In the initial moment of the development, I used this method, but then I switched to another method consisting in cloud functions integrated with the admin sdk.

Firebase SDK Firebase provides different pre-built SDKs for different platforms to interact with the Firebase services. It means that if Firebase offers the SDK for the platform in use by the developer (e.g., Android), then the developer can use directly the Firebase services through the functions written in the SDK and without making explicitly HTTP request as when using the REST API or when calling explicitly the Cloud functions. For example, the developer can access the database and authentication services using functions written in the SDK, or can call the cloud functions directly from within code always thanks to the SDK.

Firebase Admin SDK The concept is very similar to the Firebase SDK, but here the Admin SDK allows the access to Firebase services in a privileged environment. It is often used for building Server or in the cloud functions in a server-less architecture as in this thesis. For example, code running in privileged mode can access all the Firebase database and it is not blocked by the necessary restrictions that must be implemented to not expose the entire database to all the world. More on the next section.

3.1.2 Firebase security

As described before, the Firebase database can be accessed in various ways. There is the need to protect it from unwanted read and write from unauthorized users. For this purpose, Firebase gives to the developer the opportunity to define a set of rules that must be respected by the user in order to have access to the database. These rules are defined as a JSON object. Some examples below.

- these rules give full access to the database, i.e., both read and write access to all users, even not registered to the service. It means that anyone having the URL of the database can read/write it.

```
{
    "rules": {
```

```
            ".read": true,
            ".write": true
        }
    }
```

- these rules are the opposite of the first ones. The only way to read and write the database is from within a privileged environment, such using the Admin SDK initialized with the service key[2] of the developer Firebase account, strictly personal, or using the Admin SDK in the cloud functions (which anyway to be deployed needs the developer logged in his Firebase account).

```
{
    "rules": {
        ".read": false,
        ".write": false
    }
}
```

- last, with these rules, authenticated users can read and write the endpoint 'information/uid' only if their authenticated id (auth.uid) is equal to the node uid.

```
{
    "rules": {
        "information": {
            "$uid": {
                ".read": "$uid === auth.uid",
                ".write": "$uid === auth.uid"
            }
        }
    }
}
```

  The auth.uid is not passed by the user, but it is generated automatically by firebase starting from the authentication token of the user. The auth.uid generation happens each time the user tries to perform something for which he needs to be authenticated (so it does not happen only when using the database service)

[2]When a developer wants to use the Admin SDK, he has first to login into his Firebase project, generates a private key defined service key and use it for the initialization of the Admin SDK

3.2 Firebase integration in the system

In this section, I want to explain how I use the Firebase services described before in my thesis project, i.e., how I integrate the Firebase database, the Authentication and the cloud functions. It is straightforward to understand how the system uses the database, both from Android App and ComfortECU point of view. There is the need not only to exchange information between them, but also to keep stored in a trusty environment some information like the status of the car and the comfort parameters of the users, or more basically the link between the user and his car. For what concern the Firebase authentication service, it is needed for the management of the user accounts without flaws. Last, cloud functions are necessary, since I need some back-end code, that ensure and make possible the exchange of information between the car and user. The image 3.2 shows how the integration has been done in a visual manner.

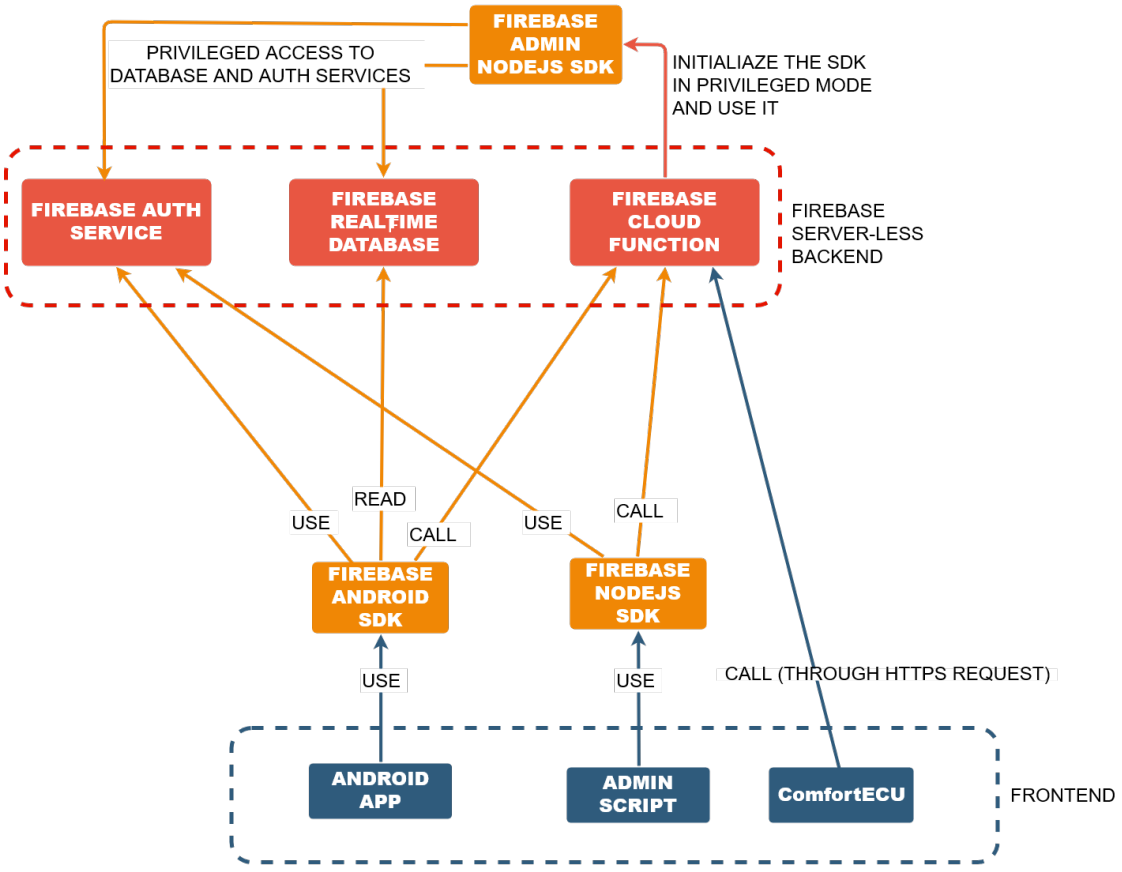

Figure 3.2. Firebase integration in the system

From now on, I will omit the term Firebase by all his services.

3.2.1 Database integration

I want to start immediately by presenting an extract of the the database JSON structure, with only one user and one car, linked together.

```
{
    "admins" : {
        "44K6QXQjJpgIyXCnorNF3KMgjSi2" : true,
        "dkW47e7rQrUoPL6O7j8FrTYCDdG3" : true
    },
    "users" : {
        "pZXmI7kE1FdnBZujAnohi8RpnVi2" : {
            "email" : "walter@comfortecu.it",
            "profiles" : {
                "pZXmI7kE1FdnBZujAnohi8RpnVi2" : {
                    "comfortset" : [ 100, 101, 102, 103, 104, 105 ],
                    "name" : "Walter"
                }
            },
            "GPS" : {
                "latitude" : "45.075443",
                "longitude" : "7.680817",
                "timestamp" : "20181027012905.000"
            },
            "carID" : "car0",
            "carStatus" : "unlock",
            "carStatusTimestamp" : 1540939791447,
            "scenarioResponse" : 1
        },
    "cars" : {
            "car0" : {
                "profileSelected" : "pZXmI7kE1FdnBZujAnohi8RpnVi2",
                "scenario" : 1,
                "scenarioTimestamp" : 1540630268896,
                "userID" : "pZXmI7kE1FdnBZujAnohi8RpnVi2"
            }
        },
        .........
}
```

It can be seen there are three important entities:

1. The admin entity under the 'admins' node composed by only the auto-genereted random unique id and a boolean . The admin has almost full access to the database through cloud functions that will be analyzed later.

2. The user entity under the 'users' node. It contains all the information related to the user like account email, car to which it is linked, the profiles connected to the user with corresponding comfort parameters and some other regarding the status of the car. To each user is assigned a unique auto-generated random id.

3. The car entity under the 'cars' node composed by all the information necessary to the car to performing the required action, like the scenario that must perform and the ID of the profile requesting that scenario. Cars have not random id assigned, but an id that must match the id hard-coded in the ComfortECU.

The database is accessed by the user through his android app but he has only reading access to his own node, while writing is delegated to the cloud functions. Instead, the ComfortECU can read and write the database only by means of cloud functions, such as the Admin user. Briefly, the database rules are represented below:

```
{
    "rules": {
        "users" : {
            "$uid" : {
                ".read" : "$uid === auth.uid"
            }
        }
    }
}
```

3.2.2 Auth integration

The user registered to the system can use cloud functions and access to his node in the database only if he is logged in the system. This is possible through the integration of the Auth service that controls automatically the identity and authentication of the user by tracking the user account life cycle and essentially by verify the identity of the user through exchange of access tokens. The steps that a user has to follow when use the system for the first time are:

1. The admin registers the user, through the Admin app, in the system with email and password. During this step, the Admin app is using the admin cloud functions.

2. The admin links the user to his car through admin cloud functions.

3. Until the user remains logged into the system, he has access to its database node (only in read mode) and to the user cloud functions.

4. When the user logs out, it is no more possible for him to control his car. Step performed through the Firebase SDK.

For what concern the admin user, he can logs into the system through a command line interface that takes always advantage of the authentication service and, only once is logged, he has the possibility to use the admin cloud functions not available to the normal user.

3.2.3 Cloud functions integration

The system uses a set of cloud functions for synchronizing car and user intents, and for store information on the database. There are:

- Cloud functions successfully invoked in the user application only if the user is logged in the system

- Cloud functions successfully invoked in the admin application only if the logged user is an admin user

- Cloud functions invoked in the ComfortECU embedded software

In practice, all cloud functions run from inside a privileged environment by using the Admin SDK, so they are able to modify and access all database nodes without being influenced by the database rules. Below, I describe the main work done by each of them.

Admin cloud functions

- `admin_createNewUser`: it gives the possibility to the admin to create a new user account, by inserting the user credential.

- `admin_fromEmailToId`: it is used by the admin for retrieving the user unique id from his email.

- `admin_insertNewCar`: the admin can insert a new car into the database by invoking this function.
- `admin_userAndCarBinding`: it is called for linking the user to his car into the database
- `admin_deleteUser`: used to delete a user from the system.

User cloud functions

- `user_createNewProfile`: when a user clicks the button to create a new profile associated to his account, this function will be triggered
- `user_deleteProfile`: used to delete one of the profile associated to the account. The user can't delete the original profile with which his account has been created. He should ask the admin to do so.
- `user_setScenario`: when the user clicks some command button on the android app, this function is invoked with the corresponding scenario that must be performed by the ComfortECU (i.e., toggle, save or apply scenario. Tracking scenario is implicitly always done by the ComfortECU)
- `user_uploadParameters`: the user can modify manually the comfort parameteres from the Android app and then upload them on the server by using this function

ComfortECU cloud functions

- `ecu_getScenarioAndSetStatus`: the ComfortECU invokes this functions in order to retrieve the scenario that must be applied and for storing information about the status of the car on database
- `ecu_setScenarioResponse`: after a scenario has been applied, the ComfortECU stores on database the result of the scenario, i.e., if it has been applied correctly or not, through this cloud function.

3.3 Algorithm & Synchronization

The system communicates by means of cloud functions that take under control the synchronization between the Android application and the ComfortECU. First, the user requests a certain scenario, the cloud function

`user_setScenario` is invoked and the requested scenario is written in the database node belonging to the car linked to the user. Second, the ComfortECU calls the cloud function `ecu_getScenarioAndSetStatus` that will first update some information related to the car, then will read the scenario to be applied, but only if the time elapsed from the scenario set and the current reading time is lesser than an established time representing the maximum network latency allowed by the system for the correct execution of a scenario. Third, if the reading is successful, the ComfortECU will perform the requested scenario and after that it will call `ecu_setSceanarioResponse` for writing the response of the scenario, but, even here, the writing is allowed only if it happens within the timing constraint imposed by the maximum network latency. In conclusion, whenever a scenario is set by the user, a time slot is assigned to this scenario for his execution. If the execution takes longer than this time slot, the eventual response is ignored and the user is warned about the impossibility to reach the car. Or, if the user tries to set another scenario before the end of the previous one or before the end of the previous time slot, the scenario set is rejected and the user, this time, is warned about the impossibility to set a new scenario if there is another still in execution. Below, all the possible cases are represented.

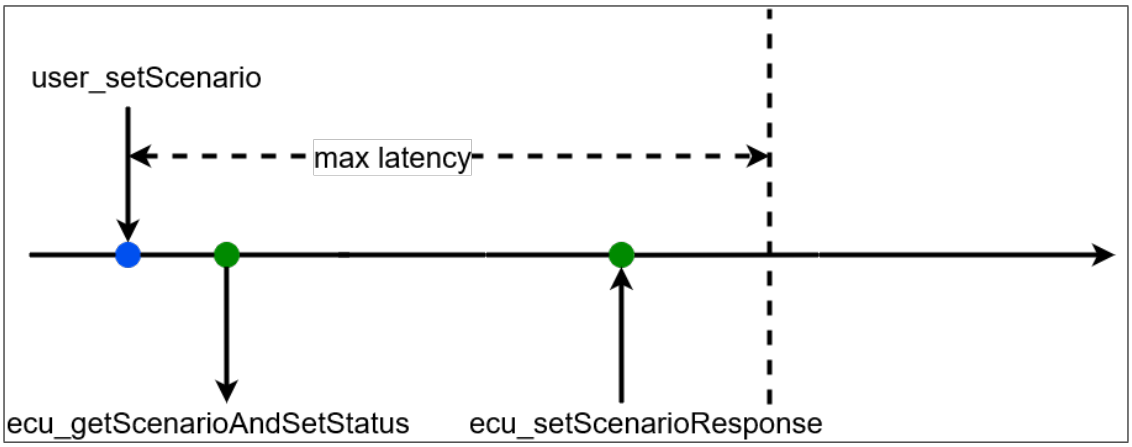

Figure 3.3. All reading and writing respect the timing constraint, so the communication is successful.

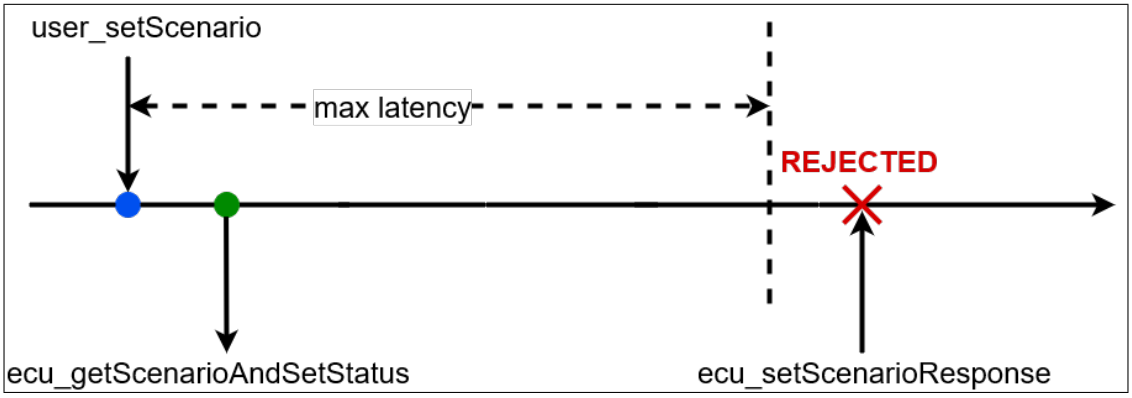

Figure 3.4. The ComfortECU tries to write too late the response of the previous requested scenario.

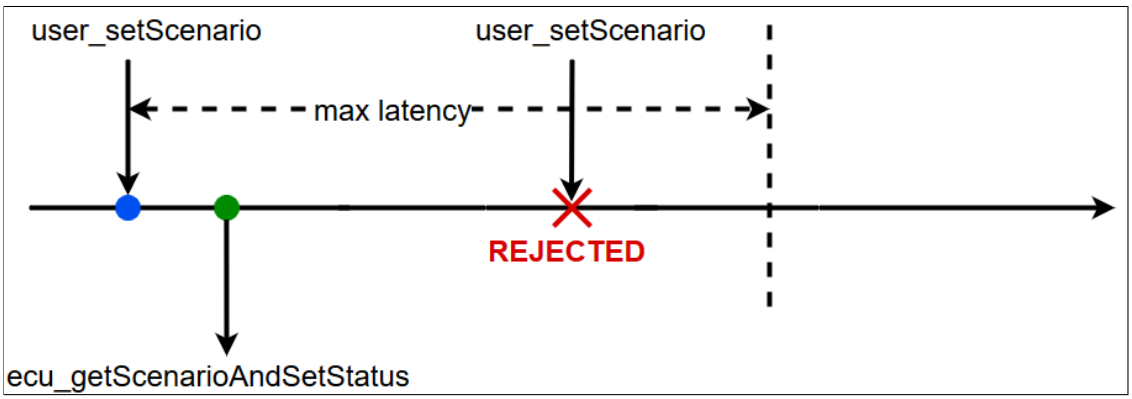

Figure 3.5. The user tries to set another scenario before the end of the previous and also before the time slot assigned to the previous scenario is expired.

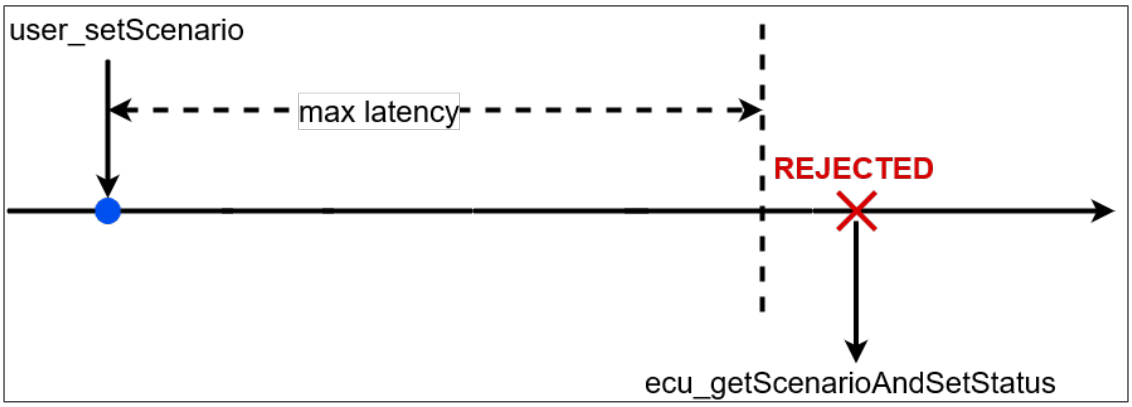

Figure 3.6. The ComfortECU tries to read too late the scenario. The reading is rejected.

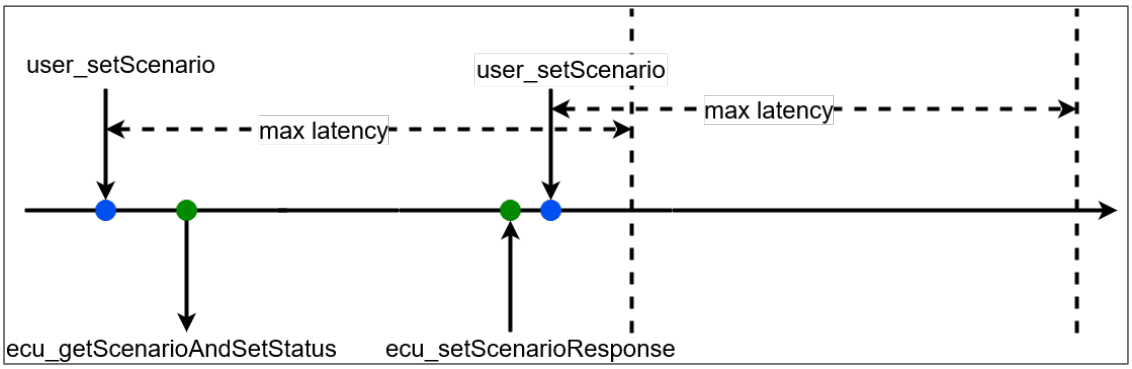

Figure 3.7. Correct communication, since the second scenario is set after the end of the previous one.

Chapter 4

ComfortECU

The ComfortECU is the component of the overall system in charge to control the status of the car (lock status and GPS position) and send CAN messages in order to perform the scenario requested by the user through his smartphone app. As described in the section 2.2.1, it is composed by:

- the *S32K144* evaluation board designed by *NXP Semiconductors*
- the *SIM808* GPRS/GSM/GPS module

The board controls the SIM808 by sending AT commands through UART communication protocol, while the interaction with the other emulated ECUs takes place via CAN protocol.
After this quick overview on the hardware configuration of the ComfortECU, it is now convenient to look also at its software composition. I can identify four main modules in the ComfortECU software:

- **FreeRTOS:** a realtime operating system that is for example used by *Tesla motors* in their vehicle gateway[13]
- **S32K14x EAR 0.8.6 SDK:** the software development kit given by NXP for the S32K14x boards, thus including the one used
- **Custom SIM808 API:** a simple API that I wrote in order to have access to the main functions offered by the SIM808
- **Server module:** a module that takes advantage of the custom SIM808 API in order to talk with the server
- **CAN module:** a module that takes advantage of the SDK in order to talk with the other emulated ECUs

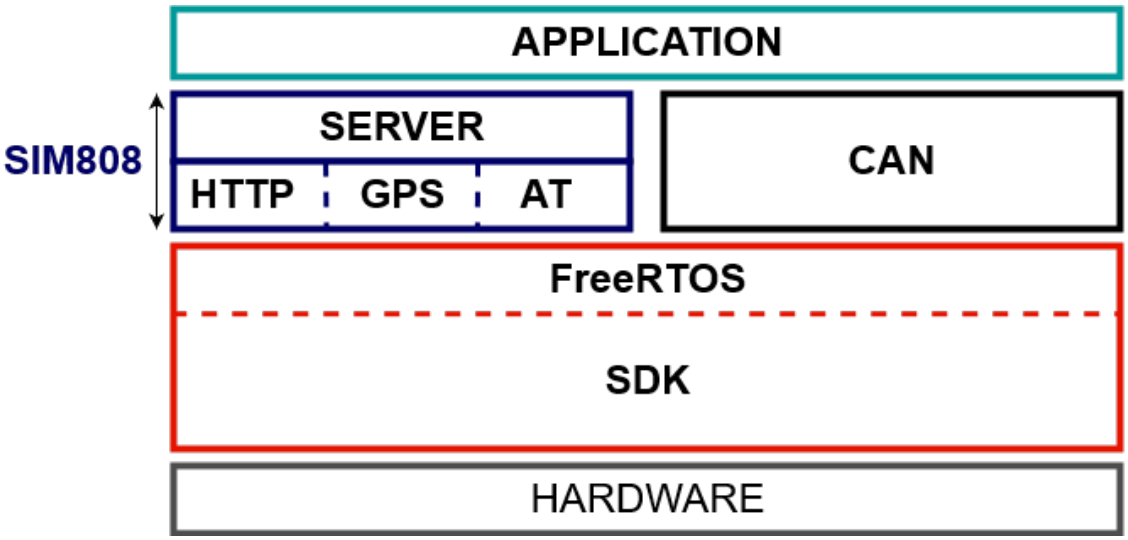

Figure 4.1. ComfortECU SW architecture

4.1 FreeRTOS

In this section, I want to describe the main characteristics of FreeRTOS. The discussion will target only the version 8 of FreeRTOS since this is the one that I use in the system. As a matter of fact, the embedded world has an immense variety of micro-controllers on which the operating system (OS) must be ported, but there are already many FreeRTOS ports ready to be used such as for the S32K144 evaluation board of the ComfortECU.

FreeRTOS architecture The core of FreeRTOS is common to all the provided ports and it is composed by only 3 files written in C language:

1. tasks.c: it has all the functions for managing tasks.

2. list.c: it provides the implementation of a linked list.

3. queue.c: it provides both queue and semaphore services.

Then, there is the device-specific portable layer in which there are files that link the core of FreeRTOS to the target platform on which the OS is running. In other words, FreeRTOS has a 'flat' architecture, i.e., the OS does not have a separation layer from the application.

Memory management FreeRTOS v8 supports only the dynamic memory allocation (from v9 has been added also the static one), therefore objects like tasks, queues, semaphores, etc., defined kernel objects, are allocated on the heap memory. It provides 5 different dynamic allocation schemes, and the developer should think carefully about the choice of the scheme since the dynamic allocation can be an issue in a real-time system. All schemes, with the exception of heap_3, allocate statically an array before the OS scheduler has been started, with a dimension specified by the developer, and use this array as heap of the application. The difference among the schemes is in how they use this array:[14]

- **heap_1:** kernel objects are allocated only before the scheduler has been started and there is no possibility to free them. The main advantages of this scheme are its determinism (the time used for the allocation is always the same) and the impossibility of fragmentation (no holes in the array).

- **heap_2:** it allows to free memory, therefore there is the need of an algorithm for deciding where kernel objects should be allocated. The algorithm is based on a best fit approach, namely, any time an object is allocated, the OS identifies which is the heap portion that best match the dimension of the object. It is obviously not deterministic (the algorithm could spend more or less time to find the best fit) and can lead to fragmentation due to the availability of the free function worsened by the impossibility to combine adjacent free blocks of memory.

- **heap_3:** as previously stated, it does not use the pre-allocated array as heap, but it lets the linker decide the allocation of the heap. It uses the standard C library "malloc" and "free" functions with all their problems in the embedded world. FreeRTOS mitigates the fact that they are not thread-safe, by temporarily suspending the scheduler.

- **heap_4:** as heap_2 but the algorithm used is based on a first fit approach and it combines adjacent free blocks of memory.

- **heap_5:** as heap_4 but the heap can be composed by more than one statically allocated array that can be placed in separated memory spaces. It can be useful in architectures with a memory block with a noncontinuous memory map.

Scheduler The application may consist of many tasks while the processor may have only one core, like in the ComfortECU, so only one task can run at a time. The role of the scheduler is to decide which of the many available tasks should run. FreeRTOS scheduler defines 4 states in which the task could be:

- **Running state:** in this state there is the task the is actually in execution
- **Ready state:** tasks that are ready to run, but they are not actually running
- **Blocked state:** tasks waiting for an event that could be either a temporal event (task is waiting for a given amount of time) or a synchronization event (interrupts, event generated by another task)
- **Suspended state:** tasks that are not available to the scheduler. For example, if the developer use blocking function without setting a finite timeout, the task enter the suspended state since it is waiting indefinitely.

Basically, the scheduler is like a callback function of the operating system that is executed in response to a periodic interrupt generated by an hardware timer of the MCU on which the RTOS is running. This interrupt is called tick interrupt in the FreeRTOS world. It is possible to use different configurations for the FreeRTOS scheduler:

1. **Fixed priority pre-emptive scheduling with time slicing:** to each task is assigned a static priority on the basis of which the scheduler decides which task should run, i.e., if a task with an higher priority than the currently running task, enters the ready state, the scheduler will force (pre-empt) the running task to the ready state in favor of the higher priority task. Instead, in case of tasks with same priority, they are executed one after the other (Round Robin) in time slots with the same period (Time slicing). One time slice is equal to the time between two tick interrupts.
2. **Fixed priority pre-emptive scheduling:** since there is not time slicing, a new ready task enters the running state only if it has an higher priority than the running task or if the running task enters either the blocked or the suspended state (no more Round Robin for tasks with same priority).

3. **Co-operative scheduling:** the running task cannot be pre-empted, so other tasks can enter the running state only if the running task explicitly enters the blocked or suspended state.

Idle task FreeRTOS defines implicitly an Idle task that run when all the other tasks are blocked or suspended. It means that the task has the lowest possible priority. It does not perform anything, but it can be used by the developer, for example, to switch the micro-controller to low power mode.

Tickless idle mode FreeRTOS allows the developer to stop the tick interrupt during idle periods without having unsynchronizations in the time maintained by the kernel. Therefore, by stopping the tick interrupt, it should be possible, in principle, to remain in low power state as much as the developer needs without synchronization problems, but this is not actually true. In fact, it is true that the tick interrupt is not generated, but, the timer connected to tick interrupt remains anyway active, and, as a consequence, when it overflows it will generate an interrupt waking up the MCU from the low-power state. Moreover, the FreeRTOS porting for S32K144 evaluation board uses as tick timer, the SysTick timer that is not able to work in low-power state, so it is not possible to transit to all low-power states provided by the MCU. The solution to these two problems is to:

- set a timer source able to run also in low power state
- set the clock frequency of the timer source to the lowest possible, such to have as less overflow interrupts as possible, and therefore remain in low power state longer

Integration in the system In the ComfortECU, FreeRTOS is configured to use heap_1 as memory allocation scheme since it is deterministic and does not suffer of fragmentation issues, while the scheduling algorithm the system uses is the Fixed priority pre-emptive scheduling without time slicing, since I want to switch to the idle task only when the main task is blocked on some event and not before. Finally, the tickless idle mode is enabled.

4.2 S32K14x SDK

The SDK provided by NXP for the development with its board offers to the developer a wide variety of features, including:

- **Device-specific header files (.h):** files containing definitions and macros for easy access to registers inside the MCU.
- **Low-level drivers:** low level drivers for each peripheral of the MCU. This layer is called hardware abstraction layer (HAL) because it abstracts the access to the registers of the peripherals wrapping the features offered by the peripheral with a set of basic functions.
- **High-level drivers:** layer of drivers built on top of the HAL. This kind of driver works at an higher level and supports the peripheral interrupt handling. Thery are called Peripheral Drivers (PDs).
- **System services:** set of software entities that are used to build the high-level drivers and can be used also directly in the application. These services are the interrupt manager (contains functions that enable/disable interrupts within the NVIC, that allow to the application to register interrupt service routine, ...), the clock manager and the power manager.
- **Middleware:** this layer makes a step forwards abstracting the High-level drivers, so providing a set of software stack that facilitate the life of the developer.
- **FreeRTOS:** the SDK has a porting of FreeRTOS that is able to uses the others SDK components (HAL, PD, middleware) thanks to a provided operating system interface (OSIF).
- **Others:** last, the SDK offers a code generator with a user interface (called Processor expert), a tool generating start-up/compiler linker files, driver examples and demos.

Integration in the system The ComfortECU uses many components of the SDK like the FlexCAN driver, the SBC middleware (for the control of the on-board CAN transceiver), FreeRTOS, and others.

4.3 Custom SIM808 API

The SIM808 module can be controlled by AT commands, i.e., commands used to communicate with modems. I wrote a simple C API to control the SIM808 in order to make more modular the whole program. The API is divided in 3 elements:

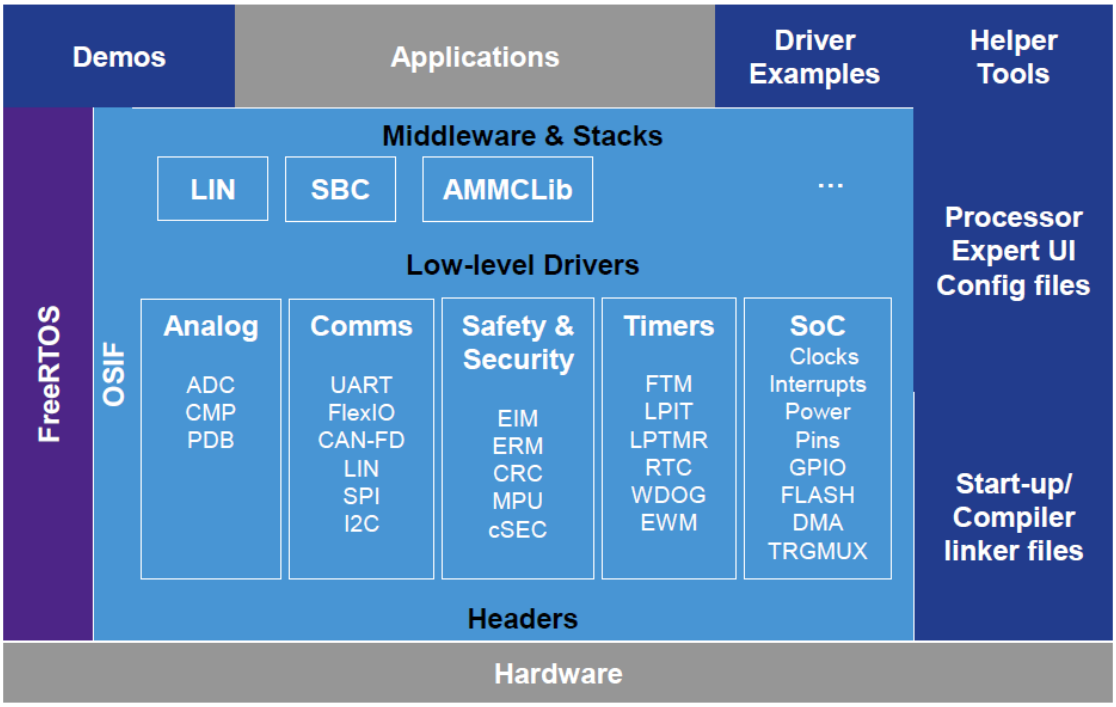

Figure 4.2. S32K14x EAR SDK

- **HTTP.c:** it has all the functions for the initiation and control of an HTTP connection
- **GPS.c:** it has all the functions for the initiation and control of the SIM808 GPS
- **AT.c:** it has the functions for parsing responses coming from SIM808 and for sending AT command

HTTP.c

- `HTTPinit`: it actually initializes the module turning on its GPRS connection.
- `HTTPsetSSL`: it is used to set or unset an SSL connection. The module comes with a preinstalled SSL certificate.
- `HTTPget`: it performs an HTTP GET request.
- `HTTPpost`: it performs an HTTP POST request.

- `HTTPdelete`: it performs an HTTP DELETE request.

GPS.c

- `GPSon`: it powers on the GPS submodule.
- `GPSoff`: it powers off the GPS submodule.
- `GPSgetPosition`: it is used for retrieving the current GPS coordinates.

AT.c

- `sendATcmd`: it is used to transmit, through UART, the AT command to the sim808 module and wait for its response. This functions depends on the micro-controller in charge to send commands, since it uses directly functions specific to its UART peripheral.
- `ATparseResponse`: it checks if the response to AT commands is equal or at least contains the expected response.
- `ATparseIPresponse`: when the GPRS connection is turned on, the SIM808 should receive an IP address. The function checks if the IP is received and saves it.
- `ATparseServerResponse`: it parses and saves the response received to a previous HTTP request.
- `ATparseGPSresponse`: it parses and saves the response received to a previous `GPSgetPosition` call.

4.4 Server and CAN modules

The server module is a C file that, as depicted at the beginning of this chapter, contains functions used to talk with the Firebase server. Basically, it is in charge to call the cloud functions deployed on the Firebase cloud (see chapter 3 for more information about Firebase and Firebase cloud functions).

- `SERVERinit`: it initializes the S32K144 hardware in order to control the SIM808 module.
- `SERVERuserInit`: it initializes the server user data structure, which is a C structure containing the ID of the user that is actually linked to the car and other information about the status of the car.

- `SERVERconnect`: it wraps the functions of SIM808 API in order to turn on the GPRS connection and set the SSL.

- `SERVERgetScenarioAndSetStatus`: it is used to call the cloud function in charge to retrieve the scenario requested by the user and to write on the database the current car status and GPS coordinates.

- `SERVERsetScenarioResponse`: it is used in order to call the cloud function in charge to write on the database the result of a previous requested scenario.

- `SERVERparseParameters`: it parses the comfort parameters received from the server.

On the other hand, the CAN module is a C file that initiates and controls the CAN peripheral of the S32K144 evaluation board. The functions included in this file are:

- `CANinit`: initiates the CAN peripheral

- `CANtransceiverInit`: initiates the on-board CAN transceiver

- `CANsetFilters`: set the CAN IDs to which the CAN peripheral must be sensitive

- `CANsend`: sends a CAN message

- `CANreceive`: prepares the peripheral to receive a CAN message

4.5 Application

The application is the main C file actuating the control algorithm. It is in charge to perform the scenario requested by the user. The task involved in all the actions is the Server Task whose flowchart is in figure 4.3.

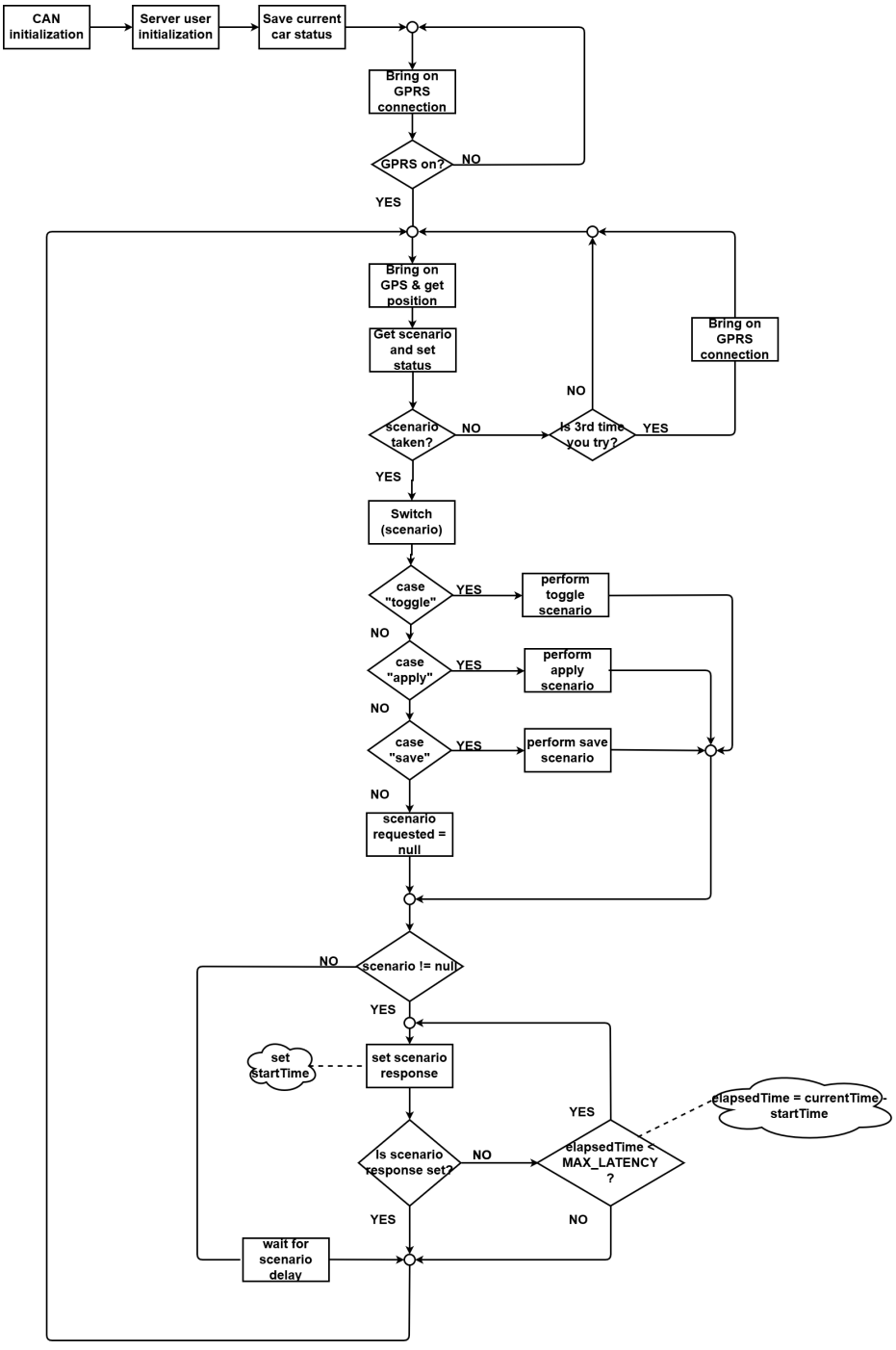

Figure 4.3. Server Task algorithm

In practice, the Server Task algorithm is divided in 3 main steps:

1. Get the scenario from the server and inform the server about the car status.
2. Perform the requested scenario.
3. Inform the server about the correct or wrong application of the scenario.

When the task completes these steps, it goes to sleep for a certain delay and then restart.

Since the flowchart does not show which are the steps performed in each scenario, the paragraphs below explain and illustrates these steps.

Toggle scenario

1. Read the car lock status
2. If the car is locked:
   2.1. send CAN messages to unlock the car
   2.2. perform the apply scenario
3. else send CAN messages to lock the car

Apply scenario

1. send CAN messages containing as payload the comfort parameters that was downloaded from the server
2. check if the comfort parameters are successfully applied in this way:
   2.1. send CAN messages requesting the comfort parameters
   2.2. receive the CAN messages containing the comfort parameters
   2.3. check if the received parameters are equal to the ones sent in step 1

Save scenario

1. send CAN messages requesting the comfort parameters
2. receive the CAN messages containing the comfort parameters
3. save the received comfort parameters in an array that will be uploaded to the server

Chapter 5

User & Admin apps

5.1 Admin app

The admin app is a simple command line interface written in Javascript that allows the system admin to perform the following actions:

- Create a new user account and insert it into the database
- Insert a new car into the database
- Link user to his car
- Delete a user account
- Create a new Admin account

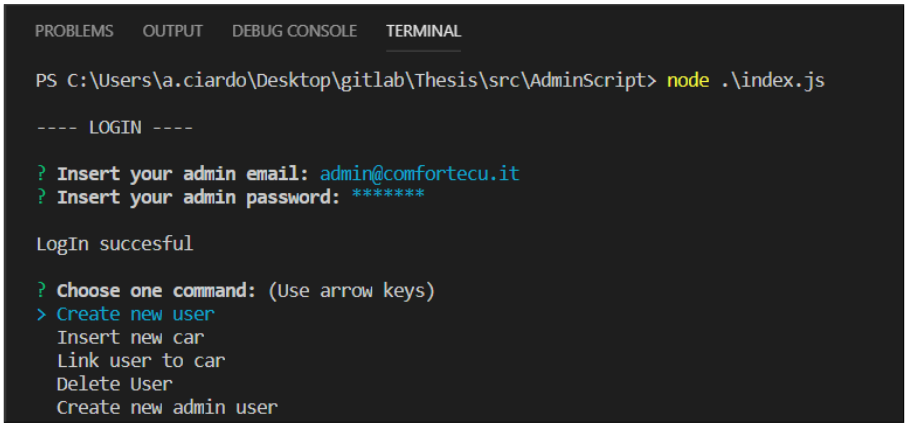

Figure 5.1. CLI view.

Normally, when a new user buys a new car, the admin should create an account associated to that user and link this account to the bought car. Later, the user can log into his Android app with the provided credentials.

5.2 User app

The user app is an Android application that allows the user send commands to the ComfortECU or to read information about the car. It is composed by two activities:

- **LogIn Activity**: it controls the user log-in
- **Main Activity**: it controls the switch among the following fragments
  - **Toggle Fragment**: from here, the user can lock/unlock his car
  - **Preference Fragment**: from here, the user can manage his preferences
  - **GPS Fragment**: from here, the user can track his car
  - **Profile Fragment**: from here, the user can manage his profiles

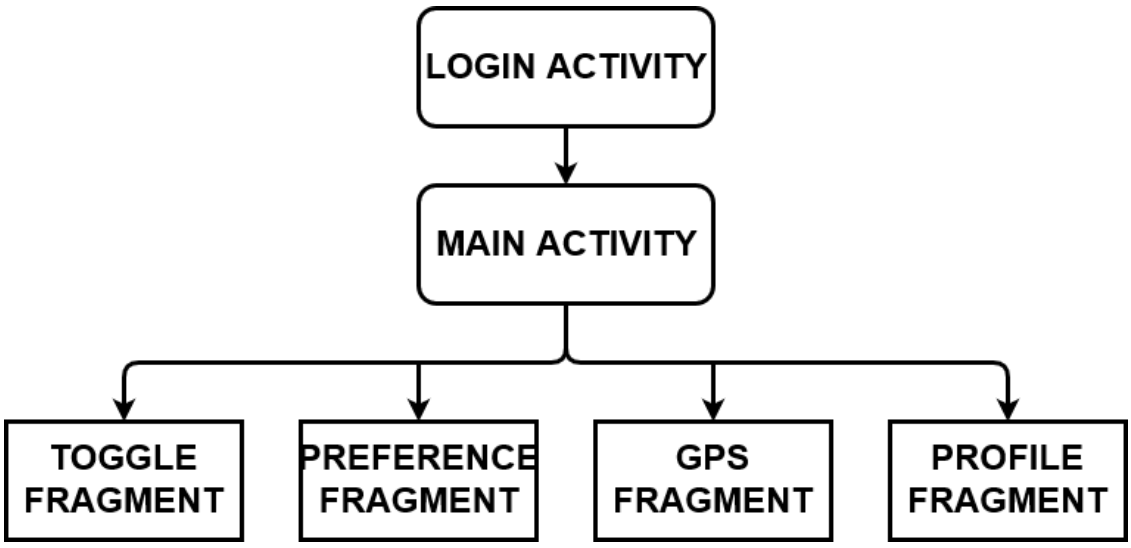

Figure 5.2. Android app architecture.

5.2.1 LogIn Activity

The LogIn activity allows the user to log into his account. The first thing the user has to insert is his email and if it is already present in the system, the user is prompt with the insertion of his password. Once the password passes the check performed by Firebase, the user can enter the app. The information about the user log-in is stored as an authentication token on the phone, so the successive times the user can enter the app without inserting again his credentials.

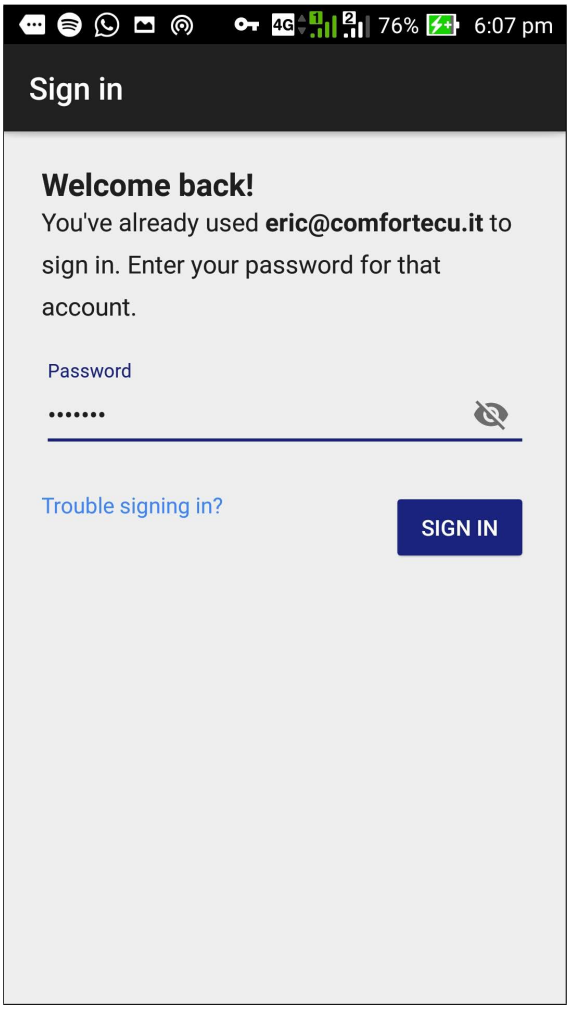

Figure 5.3. Sign In view.

5.2.2 Main activity

The main activity is launched when the user logs into the system successfully. It contains the space to accommodate a fragment and the logic to switch between different fragments. The user can choose the fragment to display by pressing the corresponding button on the bottom bar of the main activity.

Toggle fragment The Toggle fragment displays the car status (locked/unlocked) and allows the user to toggle this status, i.e., to unlock a locked car or to lock an unlocked car. The fragment is represented in figure 5.4.

Save fragment The Save fragment displays the current user's comfort preferences and allows the user to apply these preferences, or to modify and upload them, or to save them from the current car configuration. The fragment is represented in figure 5.5.

GPS fragment The GPS fragment displays the current car position on a Google map. The fragment is represented in figure 5.6.

Profiles fragment The Profiles fragment displays the profiles associated to the account and it allows the user to select another profile, to add a new profile, to remove a profile and to log-out from the system. The fragment is represented in figure 5.7.

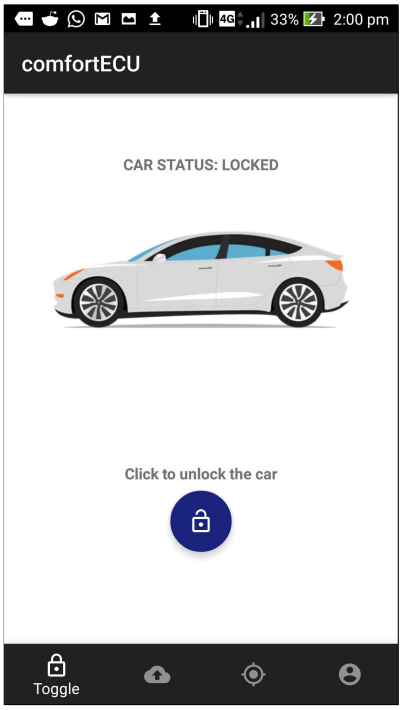

Figure 5.4. Toggle view.

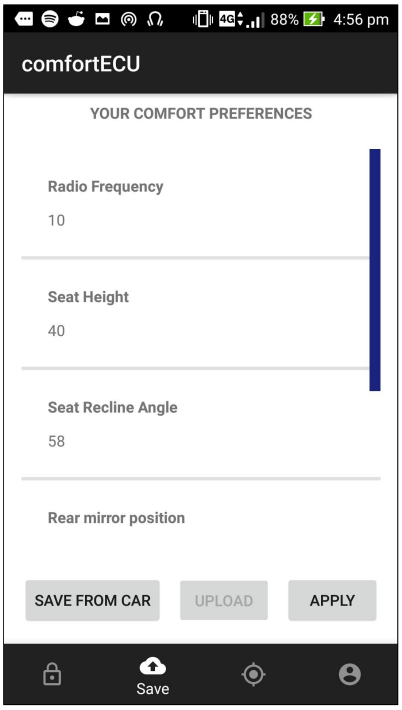

Figure 5.5. Save view.

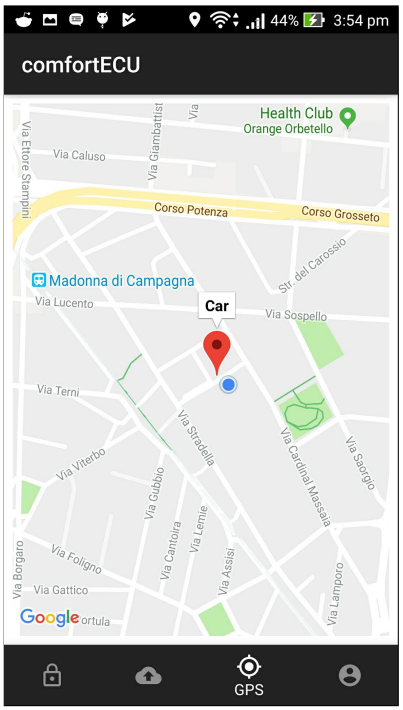

Figure 5.6. GPS view.

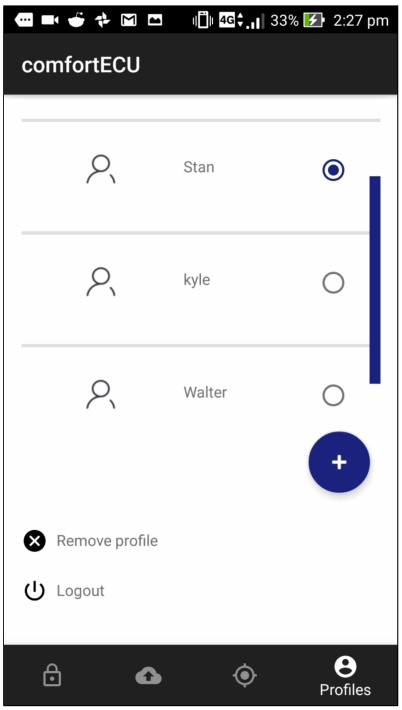

Figure 5.7. Profiles view.

Chapter 6

Conclusions

6.1 Simulation

The designed system can be view as a comprehensive system that has:

- An Android application, written in Java, that allows the user to send control commands to his car
- An ECU application, written in C, that performs the requested command
- A back-end written built with NodeJS and Firebase that connects the Android app with the ECU app.
- An admin command line interface, built with NodeJS, that gives to the Admin the control on the overall system, allowing him to create new user accounts and to connect these accounts to corresponding cars.

In this chapter, it will be shown an entire simulation of the system.

6.1.1 BUSMASTER

The ComfortECU has to exchange CAN messages with other nodes that are in the in-vehicle network. Since I did not have other real CAN nodes, I decided to emulate them on my PC by using BUSMASTER, which is an Open Source Software tool to simulate data bus systems such as CAN [15]. In order to interface the ComfortECU with the emulated nodes I used the ETAS ES581.4 USB to CAN interface module.

In-vehicle network The ComfortECU communicates with the in-vehicle network composed by the following emulated nodes:

- **lock ecu**: it controls the door and steering lock status
- **infotainment ecu**: it controls the comfort1 and comfort2 parameters
- **seat ecu**: it controls the comfort3 and comfort4 parameters
- **mirror ecu**: it controls the comfort5 parameter
- **HVAC ecu**: it controls the comfort6 parameter

CAN Database BUSMASTER allows the user to define a database of CAN messages, where it is possible to assign an ID, a payload and even a name to each CAN message. The database of the simulation is composed by the following messages:

- **set_door_lock_status**: message sent by the ComfortECU to set the door lock status
- **set_steering_lock_status**: message sent by the ComfortECU to set the steering lock status
- **get_door_lock_status**: message sent by the ComfortECU to get the current door lock status
- **get_steering_lock_status**: message sent by the ComfortECU to get the current door lock status
- **door_lock_status**: message sent by the lock_ecu containing the door lock status
- **steering_lock_status**: message sent by the lock_ecu containing the door lock status
- **comfort_set1**, ..., **comfort_set6**: : message sent by the ComfortECU to set a comfort parameter
- **comfort_get1**, ..., **comfort_get6**: message sent by the ComfortECU to get a comfort parameter
- **comfort1**, ..., **comfort6**: message sent by an emulated CAN node containing a comfort parameter

6.1.2 Simulation execution

As described in the previous chapters, the system is able to perform the following actions:

- **Toggle scenario:** The user clicks the unlock/lock button on the app, triggering the unlocking/locking of the car along with the application of the comfort parameters in case of unlocking.
- **Save scenario:** The user clicks the save button on the app in order to save the current comfort parameters configuration on the server.
- **Apply scenario:** The user clicks the apply scenario button on the app, requesting to the ECU to download from server the current comfort parameters and apply them.

There is also a tracking scenario but this does not involve the use of CAN. In the next paragraph each of these three scenarios will be simulated.

Toggle scenario

The toggle scenario depends on what is the current car lock status. If the car is locked, the toggle scenario unlocks the car and apply the comfort parameters, while if the car is unlocked, the toggle scenario just locks the car. The first case is represented in the image 6.1 where four steps are highlighted:

1. The ComfortECU send to the lock_ecu the signals to unlock both the door and the steering
2. The ComfortECU asks to the lock_ecu what is now the status of the door in order to check if the previous unlock command was executed correctly
3. The ComfortECU sends all comfort parameters, downloaded from the server, that must be set (apply scenario)
4. The ComfortECU asks to all emulated nodes to send back the current comfort parameters in order to check if the previous apply command was executed correctly

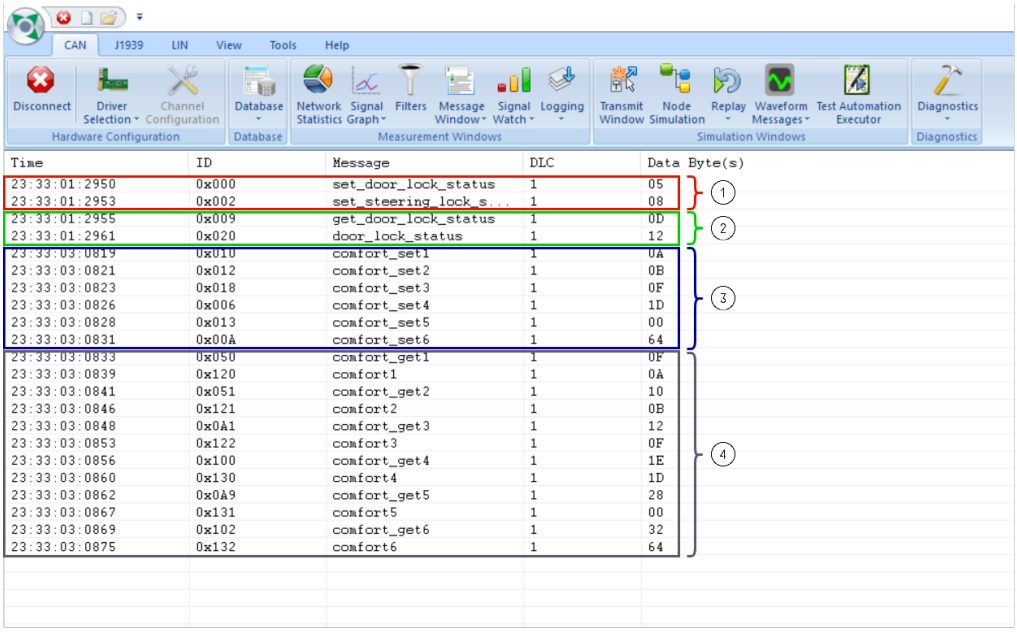

Figure 6.1. Toggle simulation: unlock

Apply scenario

The apply scenario can be divided in two steps that are shown in figure 6.2.

1. The ComfortECU sends all comfort parameters, downloaded from the server, that must be set (apply scenario).

2. The ComfortECU asks to all emulated nodes to send back the current comfort parameters in order to check if the previous apply command was executed correctly

It should be noted that the apply scenario is also a sub-step of the toggle scenario when the toggle has to perform the unlock of the car as explained in the Toggle scenario.

Save scenario

The save scenario consists only by one step in which the ComfortECU asks to each other ECU of the emulated network to send back the requested comfort parameter. Once all the parameters are collected, the ComfortECU will send them to the server.

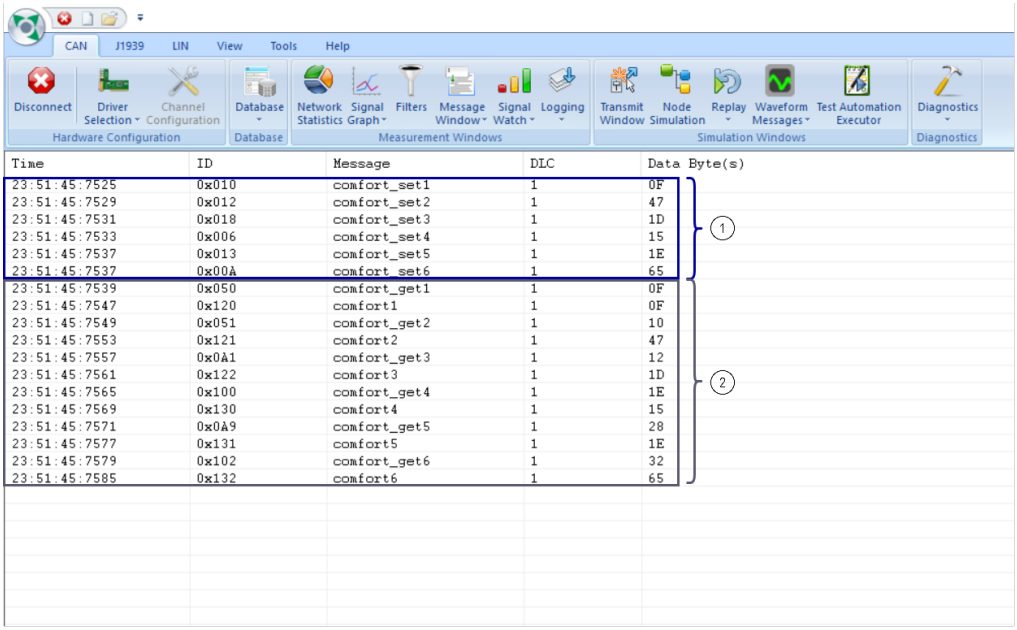

Figure 6.2. Apply simulation

Complete simulation

The figure 6.3 shows a complete simulation of the system with the following steps:

0. Lock the car.
1. Check that the lock was executed successfully.
2. Unlock the car.
3. Check that the unlock was executed successfully.
4. Send the comfort parameters that must be set.
5. Check that the comfort parameters were set successfully.
6. Save the current configuration of the car (The emulated nodes can change randomly their comfort parameters if the user press the the keyboard key 'r').

In the image is not shown the initialization phase in which the ComfortECU is started for the first time. In this phase, it controls which is the current

lock state of the car and sends this state to the server (the lock state of the car is emulated by the lock_ecu and it can be toggled by the user by clicking the keyboard key 'l').

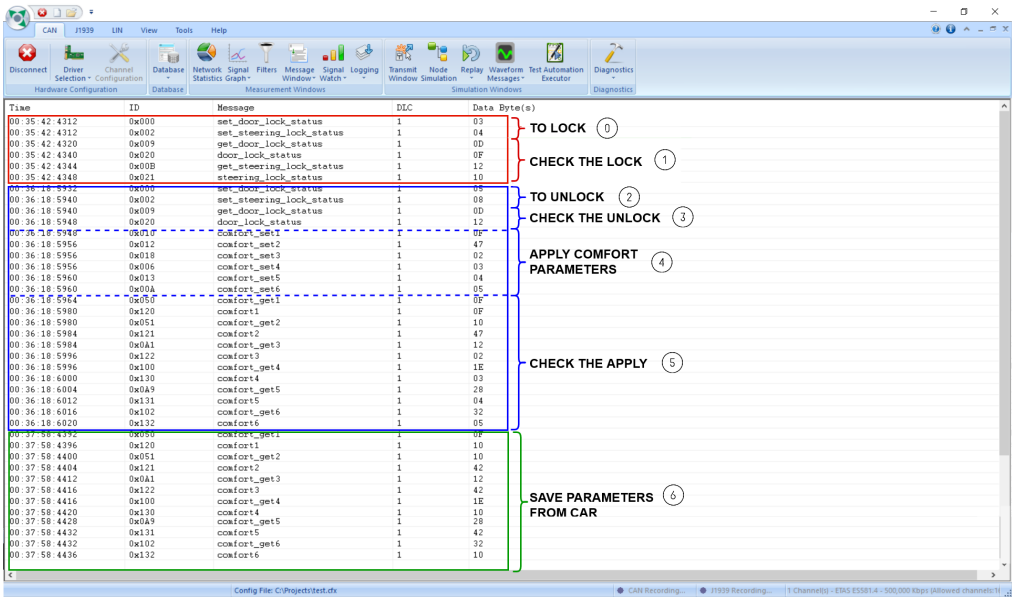

Figure 6.3. Complete simulation

6.2 Final considerations

Possible Improvements

The system can be upgraded in many ways and below there are the two main improvements that can be done:

- Use an LTE connection instead of a GPRS one in order to improve the internet connection speed
- Use a SIM card with a static and public IP in order to allow the ComfortECU to open a TCP socket and listen on that socket forever. In fact, the current system has been tested with a SIM card whose IP was NAT'd, so I was forced to implement a system that needed to register itself continuously on the server, spending more resources. There exist techniques like 'hole punching' to traverse a NAT'd network but they are not reliable and in some cases they do not work at all.

- Add another task monitoring the lock status of the system in order to detect the unlock/lock of the car also when happens by means of the car keys.

The NAT problem

I want to spend more words on NAT. NAT stays for Network Address Translation and it is a technique that dynamically assign to a private address a public address. It is also possible to map multiple private addresses to a unique public address by using different network ports. Therefore each device of a NAT'd network can be exposed on the Internet with a shared public IP address and its own port. In principle, all entities on the Internet can connect to a particular NAT'd device by issuing requests to that shared IP and that particular port, but the problem is that most ISP (Internet Service Provider) changes very dynamically all the IP addresses and ports involved in the connections. A solution could be that the NAT'd device stores its shared IP and network port on a database whenever they change so that all other entities can rely on this information to send messages to the NAT'd device. The point is that this solution is not as feasible as it would be using a static and public IP because the device needs always to contact with a certain rate a server which will detect the IP and port of the incoming connection and will update these values in the database. It is very similar to what I already implemented in my work, where I poll the server with a certain rate updating the car status and position and getting the possible scenario to be applied. So the unique true solution remains the use of a SIM card with a static and public IP which is difficult to buy as a normal customer. Usually ISPs sell this kind of solution only to business customers which buy not only one SIM card but a large quantity of SIM cards.

Conclusion

The designed system makes clear that the original idea is feasible and the development of a system that set automatically the comfort parameters of a car based on the user preferences can be implemented. The proof of concept should be obviously modified and adapted to a working environment and surely it must use a SIM card with a static and public IP, but it is anyway a good starting point for building a more sophisticated system on top of it.

Bibliography

[1] Statista, *Share of new cars sold that are connected to the internet worldwide from 2015 to 2025*, 2018 https://www.statista.com/statistics/275849/number-of-vehicles-connected-to-the-internet

[2] Avatefipour, Omid, and Hazif Malik, *State-of-the-Art Survey on In-Vehicle Network Communication (CAN-Bus) Security and Vulnerabilities.*, 5 Feb. 2018, arxiv.org/abs/1802.01725.

[3] Nicolas, Navet and Simonot-Lion, Françoise *Vehicle Fuctional Domains and Their Requirements.* Automotive Embedded Systems Handbook, 2009, CRC Press, pp. 5–5

[4] Bosch, *Central gateway CGW* www.bosch-mobility-solutions.com/en/products-and-services/passenger-cars-and-light-commercial-vehicles/connectivity-solutions/central-gateway-cgw/

[5] Stelzer, Johann, *LIN Bus–An Emerging Standard for Body Control Applications.* 24 July 2013, Electronic Design, www.electronicdesign.com/automotive/lin-bus-emerging-standard-body-control-applications.

[6] Praveen, Kollaikal, and Sridevi, Ravuri, and Eddie, Ruvinsk *Connected Cars* http://scet.berkeley.edu/wp-content/uploads/ConnCarProjectReport.pdf

[7] NXP Semiconductors, *Automotive Gateway Bridges Functional Domains and Heterogeneous Vehicle Networks* 2018, https://www.nxp.com/docs/en/white-paper/AUTOGWDEVWPUS.pdf

[8] Google, "Structure Your Database | Firebase." firebase.google.com/docs/database/rest/structure-data.

[9] Codecademy *What Is REST?* www.codecademy.com/articles/what-is-rest.

[10] Fielding, Roy Thomas. *Architectural Styles and the Design of*

Network-Based Software Architectures, 2000, www.ics.uci.edu/ fielding/pubs/dissertation/top.htm.

[11] Google, *Firebase Database REST API / Firebase.* firebase.google.com/docs/reference/rest/database/.

[12] *Introducing JSON.* www.json.org

[13] Sen, NIE, and LIU, Ling *Gateway Internals of Tesla Motors.* KEEN Lab, Tencent, 2016.zeronights.ru/wp-content/uploads/2016/12/Gateway_Internals_of_Tesla_Motors_v6.pdf

[14] *FreeRTOS - Memory Management Options for the FreeRTOS Small Footprint, Professional Grade, Real Time Kernel (Scheduler)*, www.freertos.org/a00111.html

[15] *RBEI, and ETAS. BUSMASTER*, https://rbei-etas.github.io/busmaster