Wallbox technical test

Sergi Comeche Nolla

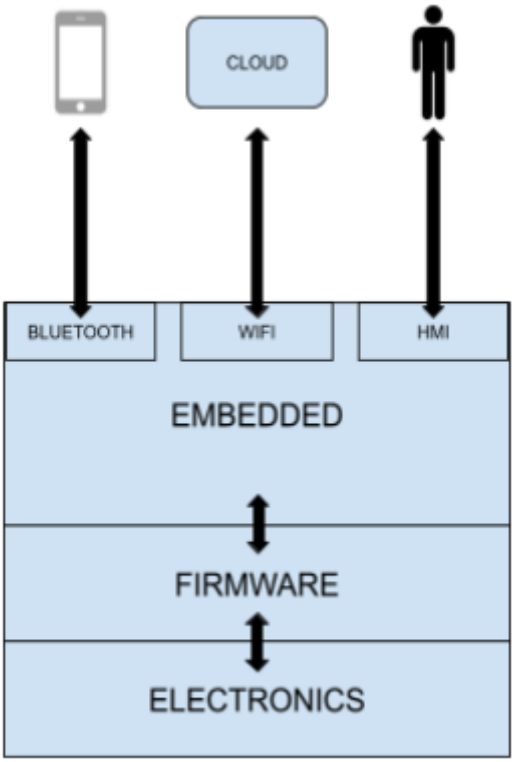
08/03/2022

Part 1: Unit Testing

You can find the link to the github repository here:
https://github.com/kmxcat/wallbox_technical_test

The three requested exercises and their corresponding unit tests are implemented and explained in github, which contains a 'readme' file with the relevant information.

Part 2: System Testing



a)

I would design a Mock Server to simulate the behavior of the embedded software in the physical device. The API would be able to point both to the physical device endpoint or the mock server, depending on the conditions of the test and if we have the actual device or not. This would be defined in a configuration file containing the IP of the desired embedded endpoint.

According to the provided figure, Cloud and Embedded communicate to each other in both ways through Wifi, so the mock must take this into consideration and include both request and response code simulations. Since Embedded and Cloud have a Wifi connection, HTTP or REST protocols could be the ones used to program the mock server.

Regarding the responses, the mock would normally be 'asleep' and wait for a request from the Cloud, which would trigger the mock response. These responses would be based on the actual code of the embedded, responding with mocked values that follow the same structure and format than real responses.

Regarding the mock requests, they can be either triggered by a user action, scheduled on a specified time or periodic to request status data.

A solid and well-known tool to use for these mocking services could be Mock-server.

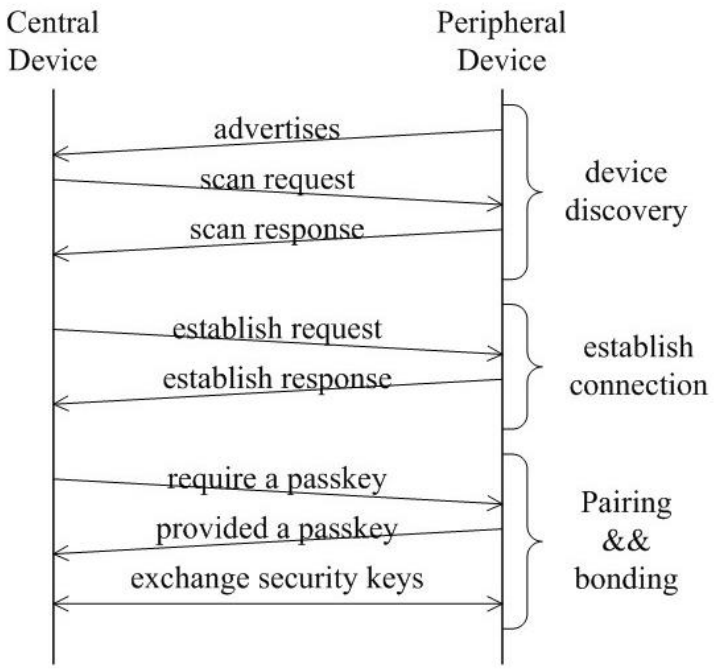
b)

Considering the data provided in the schema, the communication between the Mobile and the Embedded is performed through Bluetooth. In a case where the physical device is not available, the API should be connected to a mocked server layer containing the Bluetooth logic of the embedded software.

As it happens with the Cloud-Embedded connection, the Mobile-Embedded is bidirectional as well and the mock server would require simulating both requests and responses.

Unlike HTTP-based communications, in a very high-level Bluetooth works as follows: one device in the communication, called a peripheral, must be advertising specific pieces of data. The other device, called a central, must then be looking for these advertisements in order to detect the device sending out data. In the current situation, the peripheral could be the Bluetooth enabled physical device, while the central is a smartphone.

It is quite relevant to know that this is a continuous two-way communication between the two devices (Mobile-mock in this case). Below is a high-level diagram summarizing it:



The best approach to mock this behavior could be using Appium. It can run on real iOS and Android devices, as well as simulated mobile devices.

An interface needs to be created to aim for the mocked Bluetooth server or to the actual Embedded device, depending on the parameters given.

Finally, the most important step would be to create a class handler that implements the same interface that the real Bluetooth manager does. Doing this, when the Mobile app is communicating using the API to the Embedded software, either it is using the mocked one or the real one, it will have all the necessary methods and properties.

Achieving this would ensure that the data coming in and out the mocked Bluetooth layer is handled as expected, which is the important point regardless how it is done.