Politécnico da Guarda
Polytechnic of Guarda

Mestrado em Computação Móvel

OwnGarage: Sistema Informático para Automação de Tarefas de Manutenção em Automóveis

Luís Pedro Morgado Ribeiro

março | 2019

Escola Superior
de Tecnologia e Gestão

OwnGarage

Sistema Informático para Automação de Tarefas de Manutenção em Automóveis

Relatório de Projeto Aplicado submetido como requisito parcial para a obtenção do grau de Mestre em Computação Móvel.

Orientador: Prof. Doutor Carlos Carreto

Luís Pedro Morgado Ribeiro

Março | 2019

Agradecimentos

Agradecimento a toda a instituição do Instituto Politécnico da Guarda, seus professores e colegas de Mestrado, família e amigos.

Resumo

A indústria automóvel da atualidade presenteia-nos com automóveis cada vez mais digitais, fornecendo aos utilizadores novas funcionalidades, para que o conforto e a comodidade na condução destes seja melhorada e mais segura. Com esta digitalização torna-se importante a obtenção de dados telemétricos dos automóveis seja para a otimização de frotas e custos, quer para a deteção precoce de possíveis avarias. A inovação, não é apenas no que foi referido anteriormente, mas sim também na parte da engenharia automóvel. Isto significa que toda a engenharia que envolve a conceção dum veículo é de elevada eficiência, robustez e precisão, resultante em larga medida da evolução tecnológica atualmente presenciada. Os veículos de agora emitem menos gases poluentes, fazem melhores consumos, e os seus chassis e sistemas de segurança (quer sejam ativos ou passivos) são desenhados e desenvolvidos para proteger ao máximo todos os ocupantes do veículo em caso de sinistro. Com o desenvolvimento do sistema OwnGarage pretende-se trazer mais comodidade ao utilizador do veículo, ao utilizar um dispositivo móvel para interação com o veículo, notificando o utilizador previamente das tarefas de manutenção inerentes a este seu veículo.

Uma das partes deste sistema informático consistiu numa aplicação cliente para smartphones, a qual permitirá a recolha de dados de interesse de um automóvel com auxílio dum sistema embebido (baseado na plataforma Raspberry Pi) e que comunica com os sistemas de controlo do automóvel através do protocolo CAN (*Controller Area Network*). O sistema permite a recolha de dados de interesse de veículo, tais como velocidade, consumos, e quilometragem, permitindo que o utilizador receba proactivamente notificações relativas a tarefas de manutenção do veículo. Adicionalmente uma aplicação web permite a visualização (em forma gráfica) de alguns dados acima referidos, detalhes dos veículos e das reparações efetuadas.

Uma outra parte, foi o desenvolvimento da plataforma web direcionada para as oficinas mecânicas, a qual se tornou útil neste contexto pois permitiu uma maior eficiência e rapidez na gestão e registo das reparações automóveis.

Contudo, em relação ao sistema embebido surgiram várias dificuldades na obtenção de diversos dados devido à falta de segurança dos sistemas internos nos automóveis. Apesar destas dificuldades, os resultados obtidos nos vários testes que foram efetuados sobre o sistema revelaram-se positivos, o que significa uma boa implementação das funcionalidades desenvolvidas.

Palavras-chave Manutenção Automóvel, ODB, CAN, Raspberry Pi, Android, Angular

ABSTRACT

Current car industry presents us more with digital vehicles, which provides new functionalities to users to improve driving, safety and commodity. With digitalization, acquiring telemetric data from vehicles is an important way to optimize costs and fleets, and early detection of possible vehicle failures.

Nowadays, vehicles are more than a luxury good, becoming an integrated part in families and companies. These vehicles give their users new functionalities improving comfort and commodity in its driving. Innovation is not only on what's previously referred, but also in automotive engineering, which means that all the engineering that involves building a vehicle have a huge efficiency, robustness and precision resulting from technology evolution of today. Actual vehicles pollute less, have better fuel consumption, and their chassis and safety systems (active and passive) are designed and developed to maximize occupant's protection in case of an accident. With the development of OwnGarage is pretended to bring more commodity to users, by using a mobile device to interact with the vehicle, notifying previously this user about the maintenance tasks of his vehicle.

One part of this computer system consisted of a client application for smartphones that gathers interesting data from a vehicle with an embedded auxiliary system (based on the Raspberry Pi platform) which communicates with the vehicle control systems through CAN protocol. The system can collect data from the vehicle, such as speed, fuel consumption, mileage in a way that the user could receive proactively notifications regarding the maintenance tasks of the vehicle. In addition, a web application gives the user an interface to visualize some of this collected data (in a graph form), vehicle details and repairs donned.

Another part was the developing of a web platform directed to workshops, which becomes very useful in this context because it allowed a bigger efficiency and quickness on registering and managing vehicle repairs.

However, regarding to the embedded system we encountered several difficulties in gathering vehicle data because the lack of his internal security systems.

Despite these difficulties, the results obtained with the various tests made over the system reveals positives, which means a good implementation of the developed functionalities.

Keywords: Vehicle maintenance, ODB, CAN, Raspberry Pi, Android, Angular

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

SIGLAS E ACRÓNIMOS

AFR – *Air to fuel ratio*

BLE – *Bluetooth Low Energy*

CAN – *Controller Area Network*

CANH – CAN *High*

CANL – CAN *Low*

CI – *compression Ignition*

CSMA/CD+AMP – *Carrier Sense Multiple Access with Collision Detection + Arbitration on Message Priority*

DPF – *Diesel Particulate Filter*

ECU – *Engine Control Unit*

ED – *Engine Displacement*

EFR – *Engine Fuel Rate*

EGR – *Exhaust Gas Recirculation*

GPS – *Global Positioning Service*

HTML – *HyperText Markup Language*

IAT – *Intake Air Temperature*

IDE – *Integrated Development Environment*

IoT – *Internet of Things*

MAF – *Mass Air Flow*

MAP – *Manifold Absolute Pressure*

MM – *Molecular Mass*

MVC – *Model View Controller*

NFC – *Near Field Communication*

NOx – *Nitrogen Oxides*

OBD – *On-board Diagnostic*

R – *Gas Constant*

RFID – R*adio Frequency Identification*

RPM – *Revolutions Per Minute*

RTC – *Real time clock*

SI – *Spark Ignition*

URI – *Uniform Resource Identifier*

VE – *Volumetric Efficiency*

Wi-Fi – tecnologia rádio para redes locais sem fios com dispositivos baseados nos standards IEE 802.11

1. INTRODUÇÃO

Este capítulo aborda o enquadramento e a motivação para o desenvolvimento do sistema descrito neste relatório, definindo o problema e os objetivos e apresentando um resumo da solução proposta para resolver o problema**.** O capítulo termina com a descrição de como este relatório está organizado.

1.1. Contexto e Motivação

Este projeto enquadra-se na área da Computação Móvel, tentando, através de sistemas embebidos, integrar os automóveis com os dispositivos móveis, envolvendo tanto a Engenharia Informática como a Engenharia Automóvel. Com a maior integração dos automóveis na sociedade, este projeto surge de forma que os utilizadores dos automóveis consigam gerir simplificadamente as tarefas de manutenção, passando o dispositivo móvel a alertar para a proximidade destas tarefas e também de outras tarefas personalizadas definidas pelo utilizador.

A maior parte dos sistemas semelhantes existentes, necessitam que o sistema embebido esteja em constante comunicação com o dispositivo móvel (algo que neste projeto não é necessário) e no geral não recolhem a telemetria do automóvel. Existem também, neste momento, aplicações móveis proprietárias de várias marcas de automóveis, que já integram com os automóveis dos utilizadores, notificando o mesmo das revisões e de possíveis problemas no veículo. Contudo, estas aplicações apenas funcionam com um pequeno número de automóveis (automóveis mais recentes).

O sistema desenvolvido tem a designação de OwnGarage e funciona com automóveis que possuam interface *On-Board Diagnostic II* (OBDII) e que implementem a comunicação com o protocolo *Controller Area Network* (CAN).

Para além de possuir uma aplicação móvel, fornece também aos utilizadores um portal web para que estes possam verificar as reparações, dados e estatísticas dos seus automóveis. Este mesmo portal web tem também um outro módulo destinado às oficinas que permite centralizar e auxiliar a gestão das reparações efetuadas em cada oficina.

Tanto o portal web como aplicação móvel fazem um uso duma base de dados localizada no *backend*, que define o modelo e regras de negócio de todo o sistema.

1.2. Definição do Problema e Objetivos

Qualquer cidadão que possua um automóvel, tem de imediato conseguir gerir as tarefas de manutenção do mesmo definidas pelo plano de manutenção da marca. Embora isto não seja um problema, muitos dos cidadãos não respeitam as datas e quilómetros destas manutenções, seja por esquecimento ou por indisponibilidade do próprio cidadão. Tecnicamente não existe um sistema (exceto o automóvel) que notifique de forma pró-ativa o utilizador destas mesmas tarefas. Ao mesmo tempo, muitas das oficinas mecânicas possuem sistemas de gestão de reparações cuja base de dados é local, podendo em caso de várias filiais, existir duplicação de dados de clientes. Assim, os seguintes objetivos pretendem resolver estes problemas:

- Criar uma aplicação móvel que interaja com o automóvel, recolhendo telemetria do próprio automóvel.
- Centralizar automóveis e frotas de automóveis numa única plataforma.
- Notificar previamente os utilizadores das tarefas de manutenção programadas.
- Centralizar o registo de reparações.
- Simplificar o processo de registo de reparações.
- Melhorar a gestão das reparações, reportando tempos despendidos em cada reparação.

1.3. Solução Proposta

O sistema informático que serve de solução para o problema referido na secção anterior possui uma base de dados em MySQL no *backend*, um sistema embebido composto por: um computador Raspberry Pi, controlador CAN e um módulo RTC, que recolhe telemetria do automóvel, uma aplicação móvel desenvolvida em Android que além de transmitir os dados recolhidos pelo sistema embebido para a base de dados, também notifica os utilizadores da proximidade das tarefas de manutenção programadas, e uma aplicação web desenvolvida em Angular para que os utilizadores possam consultar e verificar também notificações relativas à proximidade das tarefas de manutenção

programadas. A aplicação web possui uma parte direcionada às oficinas, a qual centraliza toda a gestão e registo de reparações efetuadas nestas.

1.4. Organização do Relatório

Este relatório inicia-se com a motivação e descrição do contexto onde se insere o OwnGarage, definindo o problema, objetivos e a solução proposta.

De seguida, o capítulo "Trabalho Relacionado" continua com uma descrição das tecnologias e ferramentas utilizadas, e também um resumo de vários sistemas similares ao OwnGarage, enquadrados no âmbito deste.

No capítulo "Desenvolvimento do OwnGarage" é explicado como cada um dos subsistemas do OwnGarage foi implementado, segmentado pelos módulos Base de dados, Aplicação WEB, Aplicação móvel, Sistema embebido e Cálculo do consumo de combustível.

Segue-se o capítulo "Testes e Resultados" detalhando os resultados dos diferentes testes que foram efetuados sobre o OwnGarage.

Por fim, o capítulo "Conclusões", que além da conclusão, detalha também as várias funcionalidades futuras para o OwnGarage.

2. TRABALHO RELACIONADO

Este capítulo apresenta as principais tecnologias utilizadas no projeto e contém o estudo de aplicações semelhantes ao OwnGarage que contribuíram em parte para o desenvolvimento deste sistema.

2.1. Tecnologias utilizadas

O OwnGarage utiliza diversas tecnologias como os protocolos CAN e OBD especificação 2. De seguida é descrito resumidamente e respetivamente cada um destes protocolos. Estes dois protocolos eram desconhecidos e foi necessário o seu estudo para compreender o funcionamento das comunicações entre as várias unidades presentes nos automóveis, e também porque atualmente o OBD é única forma que existe de interação entre o automóvel e sistemas embebidos exteriores ao mesmo.

O protocolo CAN (BOSCH) foi desenvolvido inicialmente pela empresa BOSCH em 1983 com a primeira data de lançamento em 1986, e tinha como objetivo melhorar a qualidade dos automóveis, tornando-os assim mais rentáveis, seguros e com consumos mais eficientes. À medida que a tecnologia avançava, os automóveis começaram a tornar-se mais complexos e com maior número de dispositivos eletrónicos, que necessitavam de comunicar entre sim, fazendo com que a quantidade de cablagem no automóvel aumentasse. Assim a comunicação entre os vários dispositivos era feita ponto a ponto, tal como demonstrado na figura 2.1.

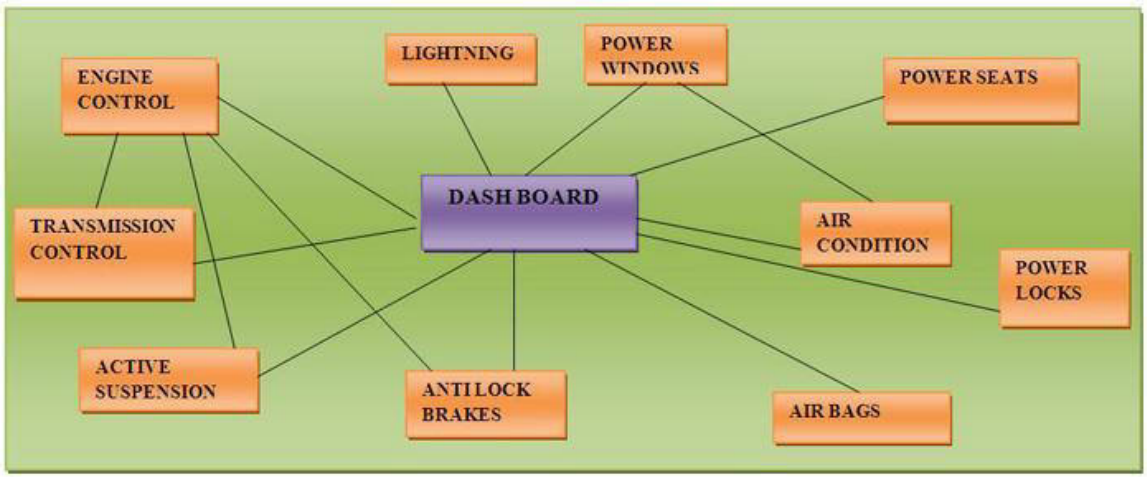


Figura 2.1 – Comunicação sem CAN, Adaptado (Parikh, 2016).

Com a implementação deste protocolo, é apenas necessário um único barramento para que todos estes dispositivos possam comunicar. A figura 2.2 demonstra a utilização de CAN na infraestrutura de comunicação dum automóvel. CAN facilita também a comunicação de múltiplos domínios (entenda-se domínio como um grupo de dispositivos que possuem requisitos similares para funcionarem no sistema). Embora tenha sido criado para satisfazer as necessidades da evolução automóvel, este protocolo é também utilizado em indústrias marítima, aeroespacial, entre outras.

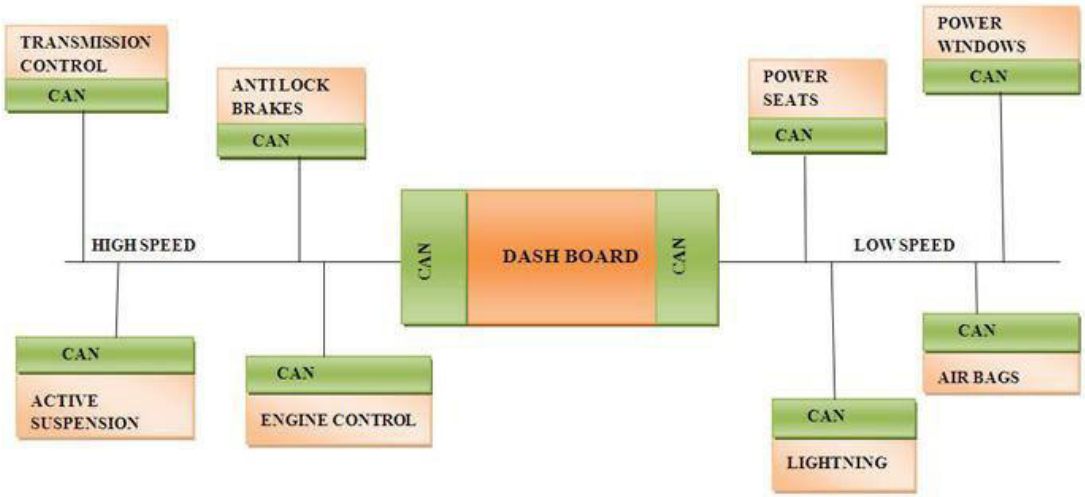


Figura 2.2 – Comunicação utilizando CAN, Adaptado (Parikh, 2016).

CAN funciona sobre um barramento em série e em *broadcast*, ou seja, todos os nós conseguem "ouvir/escutar" todas as transmissões. Logo não é possível enviar uma

mensagem para um nó específico, e por isso cada nó só reage às mensagens que são do seu interesse. Cada mensagem contém um ID que identifica a fonte ou o conteúdo da mensagem, e é através deste ID que o nó decide se processa ou descarta a mensagem.

É um protocolo de comunicação CSMA/CD+AMP (*Carrier Sense Multiple Access with Colision Detection + Arbitration on Message Priority*), o que significa que cada nó presente no barramento consegue "escutar" este bus e detetar a existência de colisões, e ainda que, existe uma "arbitragem" para o envio das mensagens (*frames* CAN) entre os nós. Esta arbitragem será explicada posteriormente.

Num barramento CAN, dois canais (CANH e CANL) coexistem lado a lado. Quando um dos canais transporta uma determinada corrente, o outro transporta uma corrente de intensidade igual, mas com sentido contrário. Uma importante característica do CAN, é que os nós ao escutar o barramento "percebem" a existência (ou não) de atividade neste. Quando o estado no barramento é "*recessive*" ou HIGH (1) lógico, o barramento não possui qualquer atividade, e durante este período ambos os canais estão tipicamente à mesma voltagem, que é aproximadamente VCC/2 (mas pode não ser se utilizarmos VCC=3.3V, como na figura 2.3). Durante o período em que o barramento se encontra ativo, o estado presente neste é o "*dominant*" ou LOW (0) lógico, e a diferença de tensão entre os dois canais é descrita pela equação (1) (Monroe, 2013):

$$\mathrm{CANH} - \mathrm{CANL} \geq 1.5\mathrm{V} \qquad (1)$$

Como um sinal "*dominant*" subscreve sempre um sinal a "*recessive*", isto permite a um nó que transmita um 1 lógico (HIGH) detetar que um outro nó está a enviar um 0 lógico (LOW) ao mesmo tempo.

Sintetizando, "*dominant*", sempre que a diferença de tensão dos dois canais (CANH-CANL) é superior a 1.5V; e "*recessive*" se esta mesma for inferior a 0.5V. Como os estados "*recessive*" é o lógico HIGH (1), e o "*dominant*" é o lógico LOW (0), segue assim a denominada lógica invertida (o maior valor da tensão representa o LOW lógico, e o menor valor da mesma o HIGH lógico).

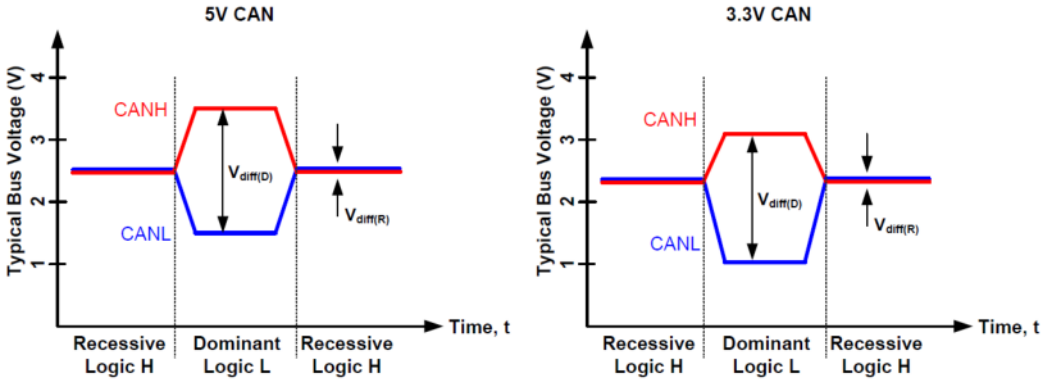


Figura 2.3 – CAN 5V vs CAN 3.3V, Adaptado (Monroe, 2013).

Antes de começar qualquer transmissão, todos os nós "escutam" o barramento de modo a detetar qualquer atividade nele. Se o sinal no barramento for "*recessive*" (1 lógico/diferença entre CANH e CANL inferior a 0.5V), o nó pode começar a transmitir. Cada *frame* que será enviada, começa sempre por um bit a 0, ou seja com o sinal "*dominant*" (diferença de tensão entre CANH e CANL superior a 1.5V). Qualquer nó que esteja escuta, perante esta tensão, percebe que um outro nó está a transmitir e por isso só deve começar a transmitir, após a transmissão da mensagem do outro nó estar completa.

Uma outra característica do CAN que torna o uso deste protocolo bastante atrativo perante um ambiente de controlo em tempo real é a alocação da prioridade das mensagens no identificador. Como o acesso ao barramento pelos nós é feito aleatoriamente, existe uma elevada probabilidade de vários nós tentarem ocupar este barramento ao mesmo tempo. Neste caso, a prioridade na transmissão dos nós envolvidos, é feita através de uma "arbitragem *bit* a *bit*". Quanto menor for o número em binário do identificador, maior será a prioridade. Um identificador que consista totalmente por zeros possui a maior prioridade na rede, visto que é a mensagem que consegue manter o bus no estado "*dominant*" (ou 0 lógico, ou voltagem maior) por maior tempo. Logo perante um cenário de duas transmissões que se iniciam em simultâneo, o que se sucede é o seguinte (e exemplificado na figura 2.4):

1. Ambos os nós iniciam as suas *frames* com o SOF (*start of frame*) bit com o valor 0 lógico ou "*dominant*".
2. Nenhum deles se apercebe que ocorreu uma colisão, e ambos mantêm-se sincronizados.

3. Ambos começam o envio do segundo campo da *frame* que é o "*11-bit Identifier*", ou seja o identificador da mensagem.
4. Como referido anteriormente, a mensagem cujo identificador possua menor valor em binário é aquela que é mais prioritária.
5. Como ambos os nós, além de estarem a enviar, estão também à escuta do barramento, aquele que possui a mensagem menos prioritária irá aperceber-se (à medida que transmite o ID da sua mensagem) que o sinal que colocou no bus não corresponde ao sinal que está a receber. E como um bit a "*dominant* (0)" subscreve sempre um bit a "*recessive* (1)", este nó percebe também que uma mensagem mais prioritária está a ser transmitida, e suspende a sua transmissão. Devido a este efeito de subscrição do sinal "*recessive*" pelo "*dominant*", o nó que possui a mensagem mais prioritária, nem se apercebe que houve uma colisão.
6. Todos os nós recebem a mensagem (visto que é feita em *broadcast*) e após o barramento entrar num estado de inatividade, é que os nós tentam uma nova transmissão dos seus dados.

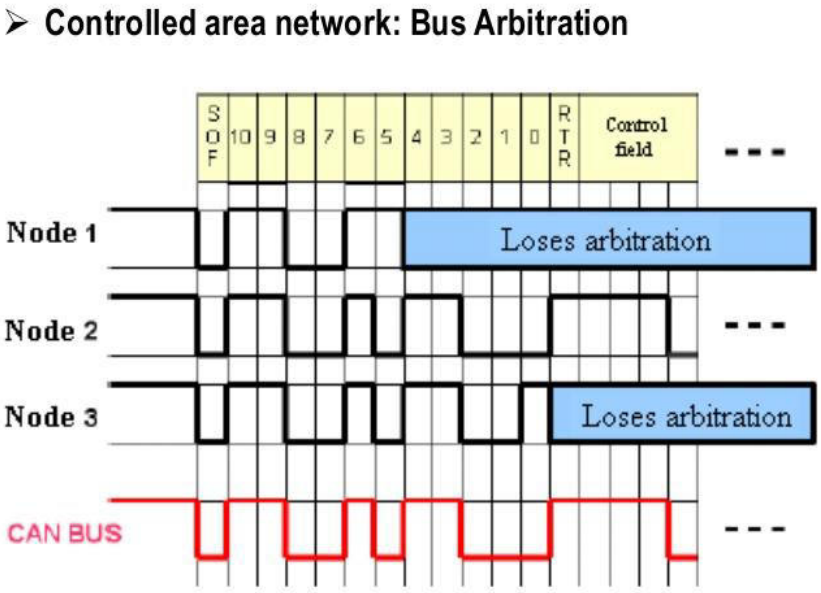


Figura 2.4 – Arbitragem CAN, Adaptado (Khazi, 2017).

A informação é transportada pelo barramento como a diferença entre dois canais (CANH e CANL). Se ambos os canais estiverem perto de uma tensão igual, bus transporta um bit a "*recessive*" (1 lógico). Mas se o canal CANH, comparado com o canal

CANL, possuir uma tensão superior a 1.5V, o bit que é transportado é "*dominant*" (0 lógico). Geralmente estes dois canais coexistem lado a lado na infraestrutura, e por isso na existência de uma interferência (ruído), é com elevada probabilidade que esta afete estes dois canais da mesma forma. Assim, mesmo com esta interferência, a diferença de tensão entre as duas linhas continua a ser igual (aproximadamente), e é possível saber qual o sinal (*dominant* (0) ou *recessive* (1)) que o barramento transporta. A especificação do protocolo CAN inclui não só as regras acima a nível da *Layer* 2 da camada do modelo OSI, mas também regras na camada física e também os diversos tipos de *frames* que cada nó pode gerar. O controlador CAN que está presente no sistema embebido tem como função gerir toda esta comunicação com o automóvel, utilizando para isso uma biblioteca em Python. Para este projeto apenas foram utilizadas *frames* do tipo "*Data Frame*" demonstrada na figura 2.5:

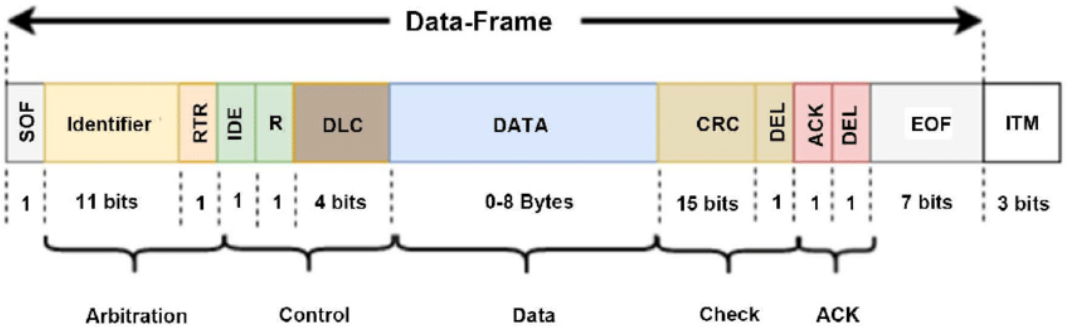


Figura 2.5 – CAN Data Frame, Adaptado (Abdulaziz Alshammari, 2018).

Quando se utiliza a biblioteca em Python deste controlador, apenas é necessário definir o "*Identifier*" e o "*Data Field*", pois o controlador por si gera os restantes campos.

Em relação ao OBD (AutoTap, 2013), este surge como um standard de diagnóstico automóvel. Desde os anos 70 e 1980 que os fabricantes utilizam a eletrónica para o controlo das funções do motor e para diagnosticar problemas, tendo também o objetivo principal do controlo de emissões de forma a que estas se mantenham nos valores definidos nos standards da *Environmental Protection Agency*. Para que estes standards fossem cumpridos, os fabricantes de automóveis desenvolveram novos sistemas de injeção controlados eletronicamente, tornando os seus sistemas mais eficientes e menos poluidores. Todos os automóveis fabricados a partir de 1996 possuem uma interface de

diagnóstico, sendo que, dependendo do ano e do fabricante, os protocolos de baixo nível (protocolos que encapsulam as mensagens OBDII) poderão ser diferentes. Para este projeto, as mensagens OBDII que possuem um formato específico serão encapsuladas numa *frame* (*layer* 2) CAN.

Ao desencapsularmos uma *frame* CAN, os dados OBDII úteis irão nos 8 bytes do campo *DATA* duma *frame* CAN. Para se compreender melhor, a figura 2.6 mostra como é gerada uma *frame* OBDII quando é pretendido recolher um determinado dado do veículo:

Identifier	Data (8 Bytes)							
ID	Nº bytes adicionais	Service	PID Code	na	na	na	na	na

Figura 2.6 – OBD Query Frame.

O campo *"Identifier*", corresponde ao "*Identifier* da *frame* CAN", que representa identificador do nó que deverá processar este pedido, sendo que no caso do OBDII o valor é 7DF e que corresponde a um endereço de *broadcast*, ou seja, múltiplos nós podem processar esta *frame*; "*Service*" que representa o modo de captura dos dados, sendo que o utlizado para este projeto foi o modo 1 que é referente à captura de dados em tempo real (mas por exemplo: o modo 3 é relativo ao diagnóstico de problemas nos sistemas de controlo de emissões e o modo 9 é relativo à recolha dos dados do próprio veículo como o VIN); e o "PID *code*" representa o dado que se pretende recolher (temperatura do motor, rotação do motor, velocidade do veículo, entre outros).

Como resposta à *frame* acima, a figura 2.7 representa a *frame* que a respetiva unidade do automóvel gera:

Identifier	Data (8 Bytes)							
ID	Nº bytes adicionais	Service	PID Code	Value	Value	Value	Value	Value

Figura 2.7 – OBD Response Frame.

O campo "*Identifier*", corresponde também ao "*Identifier da frame CAN*", podendo possuir diversos valores sendo um deles 7E8h (correspondente à ECU principal); "*Service*" e "PID *code*" com a mesma representação e valor da "*Query*

Frame". O que difere nesta *frame* são os restantes bytes que contêm os dados que são pedidos pela "*Query Frame*". Visto que o valor de cada byte varia entre 0 e 255, para saber determinados parâmetros são necessárias fórmulas matemáticas simples.

Como exemplo do que foi referido anteriormente, se se pretender obter o valor da rotação do motor, o "PID *code*" (Wikipedia, 2017) correspondente ao parâmetro *C*, a fórmula para o cálculo é a seguinte (Wikipedia, 2017):

$$C = \frac{256A + B}{4} \quad (2)$$

Em que *A* corresponde ao valor do primeiro byte (ou seja, o quarto byte do campo DATA) e *B* o valor do segundo byte (ou seja, o quinto byte do campo DATA).

Apesar deste protocolo ser um standard e a sua especificação incluir bastantes "PID *codes" standard* (Wikipedia, 2017) que permitem recolher outros dados em tempo real e de diagnóstico dos veículos (Moore, 2013), existem também "PID *codes*" que são proprietários e específicos de cada fabricante automóvel e que para se ter acesso a estes é necessário o pagamento de *royalties* aos mesmos fabricantes. Alguns destes "PIDs" proprietários são específicos a determinados componentes, como exemplo caixas de velocidade automáticas.

A tecnologia Bluetooth foi também utilizada para integração do sistema embebido com o dispositivo móvel pois esta é a forma mais simples de os interligar, além que a largura de banda disponível deste protocolo é suficiente para o envio dos dados telemétricos do automóvel. A versão do Bluetooth utilizada é a 4, mais conhecida por *Bluetooth Low Energy* (BLE).

Após o envio dos dados telemétricos do automóvel para a base de dados no *backend*, foi necessário desenvolver um portal web para que os mesmos possam ser visualizados. Este portal foi desenvolvido em Angular, que é uma *framework* para desenvolvimento de aplicações direcionadas para o *frontend* (a maior parte do processamento é feito pelo cliente e não pelo servidor) e para aplicações web *single page*. O desenvolvimento é feito na linguagem Typescript (que é uma evolução do Javascript) e com *templates* HTML, dentro do conceito de MVC. Como este portal se trata de uma

aplicação web, a utilização de Angular tornou simples e completo o desenvolvimento do mesmo.

Para o desenvolvimento da base de dados no *backend*, o motor de base de dados escolhido foi MySQL versão *Community*. MySQL é um motor de base de dados bastante conhecido e bastante robusto com suporte para vários sistemas operativos, e daí a sua escolha. Além disto, para o projeto em si, é um motor de base de dados suficiente que permite o desenvolvimento do modelo de dados e de procedimentos e funções necessárias para cumprimento das regras de negócio do OwnGarage A aplicação móvel foi desenvolvida apenas para a plataforma Android através do IDE Android Studio, visto que contém as ferramentas completas para desenvolvimento de aplicações móveis que utilizem a tecnologia de Bluetooth. Além disso, Android é o sistema operativo móvel mais popular do mundo.

O sistema embebido foi construído com recurso ao *single board computer* Raspberry Pi. O Raspberry Pi é um minicomputador com suporte para sistema operativo Linux e cujas características de hardware são suficientes para o desenvolvimento deste sistema (inclui um módulo de *ethernet* com e sem fios e um módulo *Bluetooth*).

As últimas cinco tecnologias referidas são triviais, não sendo necessária uma descrição detalhada do funcionamento das mesmas.

2.2. Soluções Existentes

Em "Cloud-Based Driver Monitoring and Vehicle Diagnostic with OBDII Telematics" (Malintha Amarasinghe, 2015), é apresentado um sistema capaz de recolher dados referentes à telemetria dum veículo, utilizando para isso a interface OBDII. Neste sistema, existe uma aplicação móvel capaz de interagir através de Bluetooth com um módulo ELM-327 (microcontrolador capaz de traduzir os dados transmitidos através da interface OBDII), de forma a recolher os diversos dados, processá-los, e enviar os mais pertinentes para uma base de dados num *backend* alojada na *Cloud* para que possam ser visualizados mais tarde num portal WEB. Alguns destes dados podem também ser mostrados ao utilizador em tempo real, mas este sistema tem a principal finalidade de monitorização do veículo e também a deteção de comportamentos na condução do veículo (por padrões de condução). Com a monitorização do veículo, este sistema poderá notificar o utilizador de variações de temperatura do motor do veículo e de falhas de outros componentes do veículo. Contudo, este sistema necessita obrigatoriamente de que o dispositivo móvel tenha sempre o Bluetooth ligado para a comunicação com o módulo ELM-327.

Além destes dados telemétricos, pode também ser importante saber a localização atual dos veículos. Esta *feature* é proposta em "Vehicular Data Acquisition System for Fleet Management Automation" (Ahmad Aljaafreh, 2011), cujo sistema em si pretende auxiliar na gestão de frotas de veículos, utilizando as tecnologias de GPS, Wi-Fi, OBDII e RFID. Cada condutor do veículo é identificado utilizando RFID, tendo o GPS a função de localização do veículo; o módulo Wi-fi para que sejam transmitidos os dados para um *webserver* no *backend*, quando este chega de novo ao parque; e o módulo OBDII para a comunicação com o veículo de forma a detetar anomalias no mesmo. Este sistema, segundo os autores, é relativamente barato, mas que não permite assim a monitorização em tempo real da frota. A plataforma web deste sistema é capaz de detetar inconsistências nas rotas dos veículos, ao comparar a rota dada pelo operador e a rota que foi registada através de GPS.

Os dados telemétricos do veículo permitem também aos utilizadores detetarem possíveis anomalias nestes, mesmo que não exista qualquer erro diagnosticado pelos sistemas de controlo no próprio. O sistema OwnGarage não foi desenvolvido para

a parte de diagnóstico automóvel, mas sim para a parte de análise de determinados parâmetros que são essenciais ao quotidiano dos utilizadores: quilometragem do veículo, velocidade e consumo de combustível. Para o cálculo do consumo de combustível, vários algoritmos estão descritos em: "Estimation of Fuel Flow for Telematics-Enabled Adaptive Fuel and Time Efficient Vehicle Routing" (Ilya Kolmanovsky, 2011), em que é explicado que o consumo de combustível não é um dos dados que o protocolo OBDII fornece. Para desenvolver algoritmos para cálculo do consumo de combustível, é necessário perceber como os motores a gasolina e diesel funcionam, perceber que sensores cada veículo possui, e se se trata de automóveis atmosféricos (sem auxílio de turbo) ou com um turbo/compressor acoplado. O sistema OwnGarage, implementa também algoritmos de cálculo de consumo de combustível baseado apenas nos factos do automóvel ser a gasolina ou a gasóleo e nos sensores MAP ou MAF que o automóvel possuiu, de forma a medir entrada de ar na admissão. Estes algoritmos irão revelar-se com erros, pois tal como explicado em "OBDII Data Logger Design for Large-Scale Deployments" (Kristian Smith, 2013), os motores não trabalham de uma forma linear, sendo que o controlo de injeção de combustível tem em conta vários sensores, tais como sensores de oxigénio (presente na maior parte dos veículos) que apenas atuam assim que é atingida uma temperatura de 316 graus Celsius, ficando até este valor o motor a funcionar numa relação ar-combustível estática. Resumidamente, enquanto a temperatura do motor nominal não é atingida, o motor funciona em *Open Loop*, o que significa que a ECU (*Engine Control Unit*) não utiliza os sensores de oxigénio para ajustar a injeção de combustível, e por isso, é injetada uma mistura enriquecida (mais combustível) para garantir o bom funcionamento do motor. Assim que a temperatura nominal é atingida, o motor trabalha em *Closed Loop*, passando a utilizar estes sensores de oxigénio, o que se traduz numa maior performance, menor consumo de combustível e menor emissão de gases poluentes.

Os motores são desenvolvidos para tentarem funcionar com a menor emissão de poluentes, ter uma performance e longevidade aceitável. É no artigo "The Usefulness of Diesel Vehicle Onboard Diagnostics (OBD) Information" (Mario Farrugia, 2016) que é referida a importância do OBD tanto no diagnóstico automóvel, como também para a compreender melhor como os sistemas de redução de emissões de poluentes funcionam. Os dois sistemas referidos tratam-se do sistema EGR (*Exhaust Gas Recirculation*) e

Diesel Particulate Filter (DPF). Para o sistema EGR, existe um sensor *NOx*, cuja comunicação é feita através de OBD, e cuja função é medir a concentração de *NOx*. *NOx* é a combinação de Azoto com Oxigénio, sendo a sua formação possível se forem atingidas temperaturas acima dos 2000 graus Kelvin. O sistema EGR reduz a formação de *NOx* ao diminuir a temperatura da combustão, reduzindo a própria entrada de oxigénio e permitindo que determinada quantidade de gases de escape seja introduzida na câmara de combustão na fase de admissão do ciclo de combustão. Assim a mistura torna-se mais pobre, pois existe ar e gases de escape, o que se traduz na diminuição da temperatura e na formação de *NOx*. Elevadas taxas de EGR são toleradas em baixas cargas de motores a gasóleo (motores CI – *Compression Ignition*), pois motores destes funcionam com relações de ar combustível altas (misturas pobres), com um excesso de oxigénio na ordem dos 15%. Importa referir que o comportamento do sistema EGR, segundo os autores, está dependente da carga do motor, da velocidade do mesmo. Uma aplicação móvel foi desenvolvida com o intuito de comunicar com o veículo quando este estivesse em movimento sendo capaz de registar os dados pretendidos e notificar os utilizadores quando está a ocorrer a regeneração do filtro de partículas (DPF).

É importante referir, que o sistema OwnGarage, utiliza também o protocolo e interface OBDII, para recolha de dados do veículo, isto porque apenas existe esta interface de comunicação com o veículo, mas que não necessita que o Bluetooth do dispositivo móvel esteja continuamente ligado, tal como é necessário em vários artigos relacionados com este tema. Notar também que cada fabricante possui sistemas de segurança próprios para o controlo de acesso ao barramento (sendo um dos *endpoints* a interface OBD2), barramento este onde são transmitidos dados do controlo do motor, e dos sistemas de segurança activa e passiva. Como protocolo de baixo nível, o sistema possui um controlador CAN, protocolo este. também utilizado em diversos sistemas mencionados anteriormente. É de salientar a importância do CAN, visto ser o protocolo utilizado nos dias de hoje e de elevada resiliência e robustez. De uma forma geral, os semelhantes projetos fazem uso de portais WEB, bases de dados e aplicações móveis pois estes fornecem uma interface *user-frendly* para os sistemas implementados. Para o âmbito do OwnGarage, o próprio cálculo do consumo de combustível poderá trazer erros se for escolhida uma fórmula mais genérica, mas que é aceitável visto o projeto ter um fim académico.

3. DESENVOLVIMENTO DO OWNGARAGE

Neste capítulo descreve-se como cada subsistema do OwnGarage foi desenvolvido, destacando-se os diagramas de arquitetura destes e tabelas com informação detalhada de funções implementadas.

3.1. Arquitetura do Sistema

A figura 3.1 representa duma forma genérica a arquitetura do sistema OwnGarage, mostrando os vários subsistemas que o constituem.

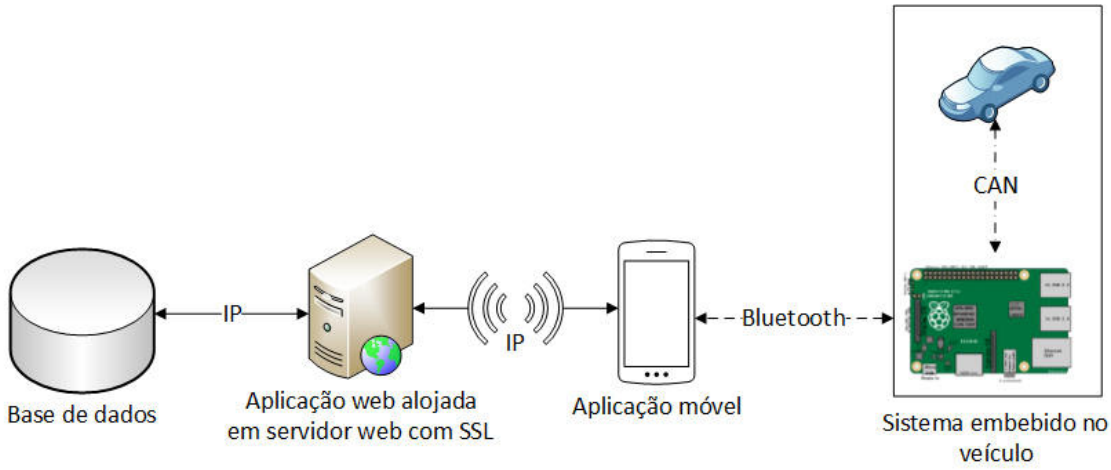


Figura 3.1 – Diagrama de arquitetura OwnGarage.

O OwnGarage é um sistema que fornece ao utilizador meios para o auxílio nas tarefas de manutenção do seu veículo automóvel, tentando também que haja uma interação entre o próprio utilizador e as oficinas mecânicas. Para que estas funcionalidades sejam atingidas, existe no *backend* uma base de dados que sustenta o modelo de negócio deste sistema. Na interface de diagnóstico do automóvel, é ligado um sistema embebido (sistema embebido no veículo) que através do protocolo CAN comunica com as unidades de controlo do automóvel. O sistema embebido recolhe telemetria do automóvel e envia-a para a aplicação móvel instalada no dispositivo móvel do utilizador, de forma a que esta mesma telemetria seja sincronizada com a base de dados no *backend*. Um servidor web contém uma aplicação web de forma que o utilizador possa

verificar esta telemetria recolhida. Também esta mesma aplicação web serve de suporte para que as oficinas registadas no sistema possam gerir e simplificar o registo de reparações automóveis.

O servidor web (que como já referido tem alojada a aplicação web) e a base de dados, estão localizadas num *backend* remoto, sendo que todas as ações que os utilizadores executam sobre o sistema são URIs invocados no servidor web, e que por sua vez estes invocam uma função específica na base de dados. A base de dados devolve o resultado da função invocada, e o servidor web encarrega-se de o transmitir à respetiva sessão do utilizador.

3.2. Base de Dados

Nesta secção descrevem-se as tabelas que constituem a base de dados, o modelo Entidade-Relacionamento (representado pela figura 3.1) e as diferentes funções e procedimentos que foram programados na base de dados. Estes procedimentos e funções são descritos sob a forma de tabela.

A base de dados foi desenvolvida de forma a ser flexível e também genérica, e cujas funções suportam as ações realizadas pelos utilizadores através da aplicação web e aplicação móvel.

O modelo de dados como foi referido anteriormente é em certos casos genérico, tendo sido desenhado para que quando haja alterações nos *frontends*, não seja necessário alterar o próprio modelo de dados. Existem campos redundantes cujo porquê será explicado no seguimento desta secção.

As tabelas desta base de dados são as seguintes (sendo descrito o seu objetivo):

- brand – guarda as marcas de veículos;
- brand_model – tabela intermédia que relaciona a marca e modelo, guardando os intervalos de quilometragem das revisões de cada modelo de veículo, de correia de distribuição e baterias nos casos aplicáveis;
- client – guarda os dados referentes a cada cliente;
- client_user – guarda a relação entre os utilizadores e clientes;
- client_vehicle – tabela intermédia que relaciona os clientes e os seus veículos;
- fuel – guarda os diferentes tipos de combustível;

- login – guarda os registos de sessão de cada utilizador (seja o acesso via Webapp ou aplicação móvel);
- model – guarda os modelos de cada marca de veículos;
- notification – guarda as quatro notificações standard que cada veículo pode possuir;
- record – guarda os registos de estatísticas de cada veículo (velocidade média, consumos, tempo, distancia);
- repair – guarda os dados de todas as reparações realizadas;
- repair_state – contém todos os possíveis estados que uma reparação pode ter;
- skill– guarda as especialidades da oficina (mecânica geral, eletricista, pneus…);
- unit – guarda as métricas que cada registo de estatística pode possuir (km/h, l/100, horas, km);
- user – guarda os registos de todos utilizadores registados;
- vehicle – guarda os dados de todos os veículos cadastrados, contendo datas e quilometragem das últimas manutenções e das próximas manutenções previstas.
- workshop – guarda os dados referentes a cada oficina;
- workshop_user – guarda a relação entre os utilizadores e oficinas;
- workshop_skill – tabela intermédia que relaciona as várias especialidades que cada oficina pode ter;

3.2.1. Diagrama Entidade-Relacionamento

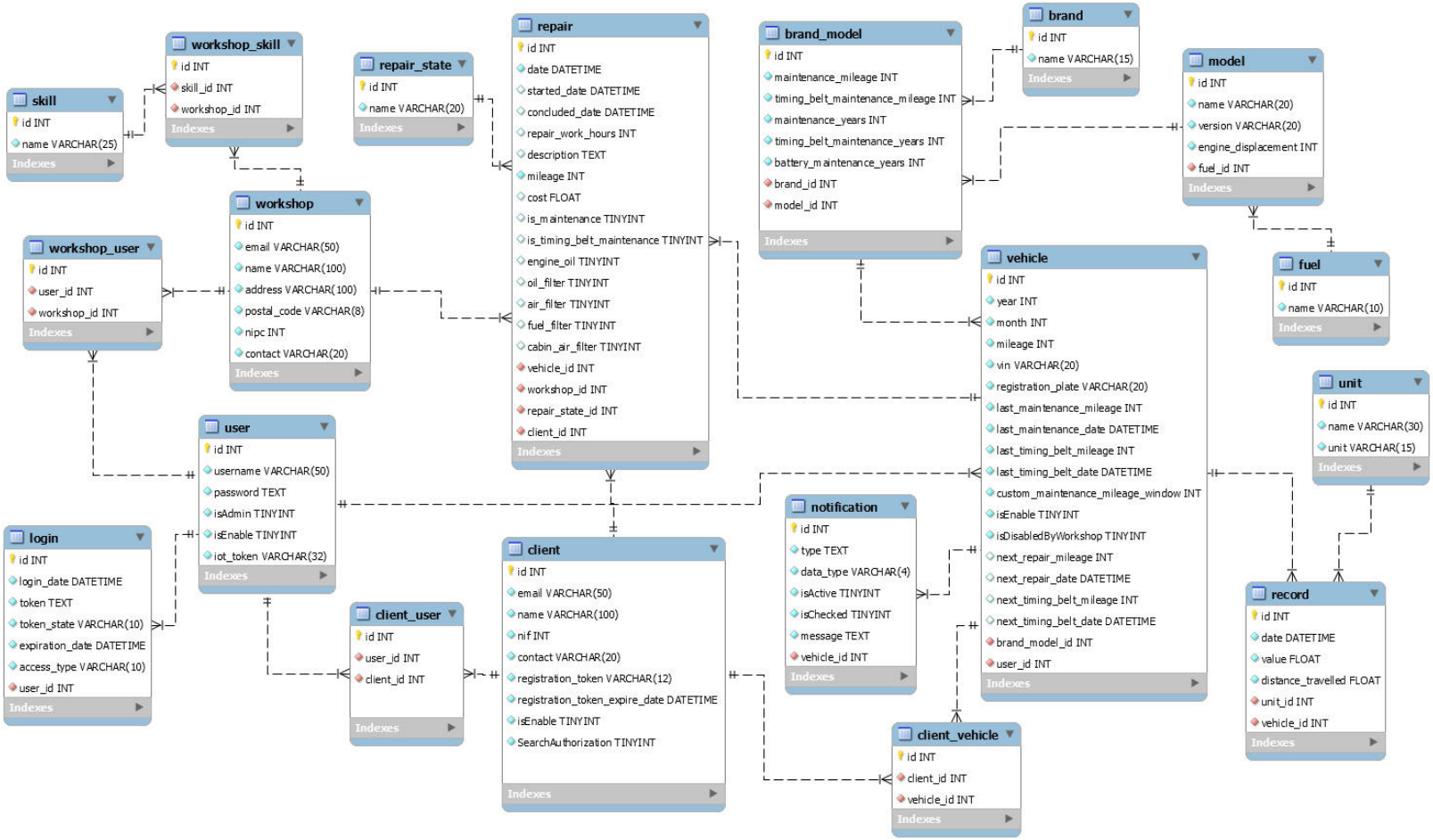


Figura 3.2 – Diagrama Entidade-Relacionamento.

3.2.2. Procedimentos e funções

A secção seguinte apresenta, em forma de tabela, as funções (e detalhes das mesmas), desenvolvidas sobre a base de dados de forma a suportar de forma correta, coerente e segura as ações que os utilizadores realizam sobre o sistema.

As três tabelas seguintes: 3.1, 3.2 e 3.3 definem três funções genéricas às duas entidades (cliente e oficina) para o processo de login e gestão das sessões.

Tabela 3.1 – Função "validate_user".

Nome:	**validate_user**
Parâmetros de entrada:	username: *username* do utilizador password: *password* do utilizador user_type: tipo de utilizador (cliente ou oficina) access_type: tipo de acesso (android ou webapp)
Parâmetros de saída/resultado:	user_id: identificador do utilizador isAdmin: define se é administrador isEnable: define se o utilizador está ainda ativo sobre o sistema token: *token* de sessão referente ao utilizador username: *username* de login
Descrição:	Valida o login de cada utilizador na aplicação. Se o utilizador for válido sobre o sistema, deverá gerar um *token* de sessão para uso desta mesma sessão, e chamar o procedimento "insert_login".
Códigos de erro	user_id = -1: utilizador inválido isEnable = -1: utilizador válido, mas inativo, não permite o login

Tabela 3.2 – Procedimento "ValidateSessionToken".

Nome:	**ValidateSessionToken**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador
Parâmetros de saída/resultado:	*True*: se sessão ativa *False*: se sessão expirada
Descrição:	Valida a sessão de qualquer utilizador. Esta função é invocada na maior parte das outras funções que alterem, adicionem ou apaguem dados da base de dados, de forma a validar a validade da sessão do utilizador a que se refere este *token* de sessão.
Códigos de erro	Não disponível

Tabela 3.3 – Procedimento "insert_loggin".

Nome:	**insert_login**
Parâmetros de entrada:	token: *token* de sessão user_id: Identificador do utilizador que fez login access_type: tipo de acesso (aplicação móvel ou Webapp) user_type: tipo de utitlizador
Parâmetros de saída/resultado:	Não disponível
Descrição:	Insere na tabela login um novo registo de forma a guardar todos os logins efetuados sobre o sistema. Cada login que seja feito via aplicação web possui um tempo de vida, sendo para os clientes de 45 minutos e para as oficinas de 10 horas. Para utilizadores da aplicação móvel, o tempo de vida da sessão é de 1 ano. É necessário para que a aplicação móvel consiga enviar dados para a BD sem que o utilizador tenha de frequentemente fazer login sobre a mesma.
Códigos de erro	Não disponível

As 5 tabelas seguintes: 3.4, 3.5, 3.6, 3.7 e 3.8 definem as cinco funções para o registo de clientes e oficinas, e também para a criação de utilizadores das duas entidades.

Tabela 3.4 – Função "register_client".

Nome:	**register_client**
Parâmetros de entrada:	email: email do cliente nome: nome do cliente nif: número de identificação fiscal do cliente contacto: contacto do cliente password: password definida pelo cliente
Parâmetros de saída/resultado:	ID do utilizador criado
Descrição:	Regista um novo cliente, e chama imediatamente a seguir a função "create_user_client". Neste modelo de dados, o cliente é também um utilizador. Posteriormente poderão ser adicionados outros utilizadores. Este utilizador registado é também administrador, podendo adicionar outros utilizadores. Esta função valida se o nome de utilizador e o NIF já existem.
Códigos de erro	-2: *username* já existente -3: NIF já existente

Tabela 3.5 – Função "create_user_client".

Nome:	**create_user_client**
Parâmetros de entrada:	email: email do cliente password: password definida pelo cliente id_client: identificador do cliente gerado na função register_client client_exists: define se é primeiro cliente/utilizador isAdmin: define se o utilizador é administrador isEnabled: define se o utilizador fica como ativo sobre o sistema
Parâmetros de saída/resultado:	ID do utilizador criado
Descrição:	Insere um novo utilizador relativo ao cliente, validando se o *username* inserido já existe.
Códigos de erro	-2: *username* já existente

Tabela 3.6 – Função "create_user_workshop".

Nome:	**create_user_workshop**
Parâmetros de entrada:	email: email da oficina password: password definida pelo utilizador id_workshop: identificador da oficina gerado na função "register_workshop" workshop_exists: define se é a primeira oficina/utilizador isAdmin: define se o utilizador é administrador isEnabled: define se o utilizador fica como ativo sobre o sistema
Parâmetros de saída/resultado:	ID do utilizador criado
Descrição:	Cria um utilizador relativo à oficina, validando se o *username* inserido já existe.
Códigos de erro	-2: *username* já existente

Tabela 3.7 – Função "register_workshop".

Nome:	**register_workshop**
Parâmetros de entrada:	email: email da oficina name: nome da oficina nipc: NIPC da oficina contact: contacto da oficina password: password definida pelo utilizador address: morada da oficina postal_code: código postal da oficina
Parâmetros de saída/resultado:	ID do utilizador criado
Descrição:	Insere uma nova oficina. Tal como na função "register_client", esta função executa imediatamente a seguir a função "create_user_workshop" para criar um utilizador. Valida se o NIPC e o *username* já existem.
Códigos de erro	-2: *username* já existente -3: NIPC já existente

Tabela 3.8 – Função "create_user_for_registered_client".

Nome:	**create_user_for_registered_client**
Parâmetros de entrada:	email: email do utilizador nif: NIF do utilizador password: password para o registo código: código para registo do utilizador quando o cliente já existe (fornecido pela oficina)
Parâmetros de saída/resultado:	Valor maior que 0, utilizador registado com sucesso e associado ao cliente
Descrição:	A oficina após registar um novo cliente, fornece o código a este para que este se possa registar na plataforma e verificar os seus veículos.
Códigos de erro	-1: código expirado -2: *username* introduzido já existente -3: código inserido errado -4: NIF não existe ou utilizador já esta registado

As duas seguintes tabelas: 3.9 e 3.10 definem as duas funções automáticas que são executadas na base de dados em *background* de forma a validarem o estado das sessões e notificações do cliente para cada veículo. Estas duas funções, ao contrário de todas as restantes, não estão expostas para serem invocadas através da aplicação web, sendo apenas invocadas pela base de dados.

Tabela 3.9 – Procedimento "update_token_session_time".

Nome:	**update_token_session_time**
Parâmetros de entrada:	Não disponível
Parâmetros de saída/resultado:	Não disponível
Descrição:	Atualiza o estado do *token*, validando se a hora atual é maior que a hora de prescrição do *token* de sessão.
Códigos de erro	Não disponível

Tabela 3.10 – Procedimento "CheckNotifications".

Nome:	**CheckNotifications**
Parâmetros de entrada:	Não disponível
Parâmetros de saída/resultado:	Não disponível
Descrição:	Esta função tem como objetivo atualizar o estado das notificações de cada veículo. Apenas altera o estado destas para ativo para que possam aparecer no *dashboard* do utilizador. Esta função é executada sobre ordem dum evento agendado para todos os dias.
Códigos de erro	Não disponível

As três tabelas que se seguem: 3.1, 3.12 e 3.13 definem funções destinadas ao cliente para a gestão dos seus veículos.

Tabela 3.11 – Função "GetVehicles".

Nome:	**GetVehicles**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador
Parâmetros de saída/resultado:	Lista de veículos
Descrição:	Devolve a lista de todos os veículos associados ao cliente e que estão marcados como ativos
Códigos de erro	-1: sessão expirada

Tabela 3.12 – Função "DeleteVehicle".

Nome:	**DeleteVehicle**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador ID do veículo
Parâmetros de saída/resultado:	ID do veículo que colocou como inativo
Descrição:	Coloca o veículo como inativo
Códigos de erro	-1: sessão expirada

Tabela 3.13 – Função "insert_vehicle".

Nome:	**insert_vehicle**
Parâmetros de entrada:	Todo os dados abaixo referem-se às características do veículo do utilizador: ano mês quilometragem vin: *Vehicle Identification Number* matrícula quilometragem da última revisão data da última revisão quilometragem da última mudança da correia de distribuição data da última mudança da correia de distribuição ID da marca: identificador da marca do veículo ID do modelo: identificador do modelo do veículo token: *token* de sessão referente ao utilizador
Parâmetros de saída/resultado:	ID do veículo criado
Descrição:	Insere um novo veículo com os dados que o utilizador introduziu. Esta função é invocada tanto na aplicação web como na aplicação móvel
Códigos de erro	-1: sessão expirada

As seis tabelas seguintes: 3.14, 3.15, 3.16, 3.17, 3.18, 3.19 definem as funções destinadas à oficina para a gestão dos clientes e seus veículos.

Tabela 3.14 – Função "GetClientHisVehicles".

Nome:	**GetClientHisVehicles**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador NIF do cliente
Parâmetros de saída/resultado:	Cliente e lista dos respetivos veículos
Descrição:	Devolve o cliente cujo NIF seja igual ao parâmetro de entrada, e os seus veículos associados. Esta função é utilizada quando a oficina pretende criar uma nova reparação.
Códigos de erro	-1: sessão expirada -2: cliente não encontrado -3: cliente existe, mas não tem nenhum carro ativo

Tabela 3.15 – Função "GetVehiclesClient".

Nome:	**GetVehiclesClient**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador Matrícula do veículo
Parâmetros de saída/resultado:	Lista de veículos
Descrição:	Devolve os veículos cuja matrícula é igual à do parâmetro de entrada, e que estejam definidos como "Ativos", tanto para o cliente, como para a oficina (a oficina pode desativar da pesquisa de veículos, veículos que estejam incoerentes com a realidade)
Códigos de erro	-1: sessão expirada -2: Veículo não encontrado ou inativo

Tabela 3.16 – Função "RegisterNewClientExistingVehicle".

Nome:	**RegisterNewClientExistingVehicle**
Parâmetros de entrada:	contacto do cliente nome do cliente NIF do cliente código para o registo do cliente token: *token* de sessão referente ao utilizador ID do veículo
Parâmetros de saída/resultado:	ID do cliente registado
Descrição:	A oficina pode inserir um novo cliente associado a um determinado veículo para lançar a factura num cliente diferente do qual o veículo possui inicialmente. Esta função valida se o NIF que a oficina introduziu já existe no sistema, e valida se este mesmo NIF (caso já exista) já está associado ao veículo que a oficina deseja criar a reparação.
Códigos de erro	-2: cliente já existe no sistema e o veículo já está associado ao cliente -3: cliente já existe no sistema

Tabela 3.17 – Função "WorkshopCreateClientVehicle".

Nome:	**WorkshopCreateClientVehicle**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador ID marca: Identificador da marca ID Modelo: Identificador do modelo Quilometragem Matrícula Ano Mês ID do cliente: Identificador do cliente
Parâmetros de saída/resultado:	1, e dados do veículo registado
Descrição:	A oficina após a pesquisa do cliente pode criar um novo veículo e associá-lo a este cliente
Códigos de erro	-1: se sessão expirada

Tabela 3.18 – Função "DeleteRegisteredClientByWorkshop".

Nome:	**DeleteRegisteredClientByWorkshop**
Parâmetros de entrada:	ID do veículo: identificador do veículo ID do cliente: identificador do cliente token: *token* de sessão referente ao utilizador
Parâmetros de saída/resultado:	1
Descrição:	A oficina pode associar um cliente ao veículo errado, ou inserir dados do cliente errados. Esta função elimina este cliente e elimina a sua relação com o veículo.
Códigos de erro	-1: sessão expirada

Tabela 3.19 – Função "DeleteVehicleWorkshop".

Nome:	**DeleteVehicleWorkshop**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador ID do veículo: Identificador do veículo
Parâmetros de saída/resultado:	1, se veículo eliminado
Descrição:	A oficina pode apagar os veículos caso estes apareçam repetidos. Neste caso, apagar significa que este veículo não irá aparecer mais nas pesquisas de veículos que as oficinas fizerem, ficando como inativo na base de dados.
Códigos de erro	-1: se sessão expirada

As quatro tabelas que se seguem: 3.20, 3.21, 3.22, 3.23 definem funções da oficina para gestão das reparações.

Tabela 3.20 – Função "RegisterNewRepair".

Nome:	**RegisterNewRepair**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador ID do cliente: Identificador do cliente Id do veículo: Identificador do veículo quilometragem: Quilometragem do veículo
Parâmetros de saída/resultado:	Valor maior que 0, reparação regista com sucesso
Descrição:	Esta função tem como objectivo o registo de uma nova reparação por parte da oficina
Códigos de erro	-1: *token* de sessão expirado -2: quilometragem introduzida pela oficina inferior à quilometragem do veículo

Tabela 3.21 – Função "GetWorkshopRepairs".

Nome:	**GetWorkshopRepairs**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador campo de pesquisa texto a pesquisar
Parâmetros de saída/resultado:	Lista de todas as reparações
Descrição:	Esta função tem como objetivo devolver a lista de todas as reparações que foram efetuadas na oficina. A oficina deverá pesquisar por NIF ou matrícula do veículo.
Códigos de erro	-1: se sessão expirada -2: se não forem encontrados resultados

Tabela 3.22 – Função "GetWorkshopDashboard".

Nome:	**GetWorkshopDashboard**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador
Parâmetros de saída/resultado:	Lista de reparações ainda por concluir
Descrição:	Esta função tem como objetivo devolver a lista de todas as reparações da respetiva oficina, mas cujo estado não seja nem "Concluído" nem "Cancelado". Os dados serão mostrados no *dashboard* da oficina.
Códigos de erro	-1: se sessão expirada -2: se não existirem reparações em aberto

Tabela 3.23 – Função "WorkshopUpdateRepair".

Nome:	**WorkshopUpdateRepair**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador ID da reparação Estado da reparação Custo Manutenção Correia de distribuição Óleo do motor Filtro de óleo Filtro de ar Filtro de combustível Filtro de habitáculo
Parâmetros de saída/resultado:	1 se reparação atualizada
Descrição:	Esta função tem como objetivo atualizar o estado das reparações. Atualiza também a quilometragem do veículo e também os campos referentes à última manutenção programada ou mudança da correia de distribuição (caso seja indicado).
Códigos de erro	-1: se sessão expirada

As quatro tabelas seguintes: 3.24, 3.25, 3.26 e 3.27 definem funções do cliente para atualização e verificação das notificações e reparações respetivamente.

Tabela 3.24 – Função "GetClientNotifications".

Nome:	**GetClientNotifications**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador "*Flag*" de administrador
Parâmetros de saída/resultado:	1 e respetivas notificações, se existirem notificações ativas
Descrição:	Esta função tem como objetivo devolver todas as notificações que estão ativas referentes ao veículo do utilizador.
Códigos de erro	-1 se sessão expirada -2 se não existirem notificações ativas

Tabela 3.25 – Função "UpdateNotification".

Nome:	**UpdateNotification**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador ID – identificador da notificação "*Flag*" de "ativa" "*Flag"* de "vista"
Parâmetros de saída/resultado:	1, se for atualizada a notificação
Descrição:	Esta função tem como objetivo atualizar o estado das notificações do utilizador, seja para as colocar como "vistas" ou como "inativas".
Códigos de erro	-1 se sessão expirada

Tabela 3.26 – Função "GetClientOpenRepairs".

Nome:	**GetClientOpenRepairs**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador
Parâmetros de saída/resultado:	Reparações em aberto
Descrição:	Esta função tem como objetivo devolver todas as reparações que estão em "aberto" ou "em progresso" do respetivo cliente
Códigos de erro	-1 se sessão expirada -2 senão existirem reparações em aberto

Tabela 3.27 – Função "GetClientRepairs".

Nome:	**GetClientRepairs**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador Parâmetro de pesquisa Limite de resultados
Parâmetros de saída/resultado:	Todas as reparações que o cliente pesquisou
Descrição:	Esta função tem como objetivo devolver todas as reparações referentes ao veículo que o cliente pesquisou, podendo escolher o limite de resultados que quer visualizar (por omissão 25, e no máximo 100)
Códigos de erro	-1 se sessão expirada -2 senão existirem reparações

As três últimas tabelas: 3.28, 3.29 e 3.30 definem funções do cliente para o envio de estatísticas do veículo e visualização das mesmas.

Tabela 3.28 – Função "SendStatistics".

Nome:	**SendStatistics**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador Estatísticas em formato JSON
Parâmetros de saída/resultado:	Número de estatísticas inseridas na base de dados
Descrição:	Esta função tem como objetivo inserir na base de dados as estatísticas que o sistema embebido recolheu e que de seguida enviou para a aplicação móvel.
Códigos de erro	-1 se sessão expirada

Tabela 3.29 – Função "StatisticsGetVehicles"

Nome:	**StatisticsGetVehicles**
Parâmetros de entrada:	token: token de sessão
Parâmetros de saída/resultado:	Veículos do utilizador
Descrição:	Esta função tem como objetivo devolver os veículos que o utilizador possui ativos para posteriormente selecionar qual pretende visualizar as estatísticas
Códigos de erro	-1 se sessão expirada -2 se o cliente não possuir veículos ativos

Tabela 3.30 – Função "GetStatistics".

Nome:	**GetStatistics**
Parâmetros de entrada:	token: *token* de sessão referente ao utilizador ID do veículo Métrica
Parâmetros de saída/resultado:	Estatísticas do veículo selecionado
Descrição:	Esta função tem como objetivo devolver as estatísticas referentes ao veículo e métrica escolhida
Códigos de erro	-1 se sessão expirada -2 se o veículo não possuir estatísticas

3.2.3. Eventos agendados

A seguinte secção descreve o funcionamento dos dois eventos cujas funções são invocadas automaticamente pela base de dados, sendo um deles minuto a minuto e o outro uma vez por dia durante a noite (Stack Overflow, 2010).

3.2.3.1. Evento "update_*token*_session_time"

Foi necessário criar um evento que seja executado a cada minuto, e com a função de invocar o procedimento "update_token_session_time", responsável por verificar a validade de cada *token* de sessão que esteja ativo. Esta validação representa um nível de segurança acrescido no acesso dos utilizadores ao sistema, porque evita possíveis ataques de terceiros que tentem utilizar o *token* de sessão dum utilizador válido. Contudo, para que estes eventos possam ser executados é necessário executar o seguinte comando como DBA na base de dados (para tal foi utilizada a conta de "*root*") (Oracle Corporation, 2019):

"SET GLOBAL event_scheduler = ON;"

O evento "update_session_*token*s" foi implementado com o seguinte código.

```
CREATE EVENT update_session_tokens
ON SCHEDULE
EVERY 1 minute
COMMENT 'Update login session tokens'
DO
call update_token_session_time;
```

3.2.3.2. Evento "CheckNotifications"

Foi necessário criar um outro evento que seja executado uma vez por dia, cuja função é atualizar as notificações dos veículos referentes às revisões e mudança da correia de distribuição de todos os veículos. Este evento deverá validar a quilometragem atual dos veículos e a data atual, e colocar as notificações como ativas nos seguintes casos:

- Quilometragem atual do veículo com diferença de menos de 1000 Km da próxima revisão do mesmo veículo
- Data atual, com diferença de um mês, em relação à data prevista da próxima revisão

- Quilometragem atual do veículo com diferença de menos de 1000 Km da próxima mudança de correia de distribuição do mesmo veículo
- Data atual, com diferença de um mês, em relação à data prevista da próxima mudança de correia de distribuição

O modelo de dados foi alterado em parte devido à execução desta função, contendo 4 colunas redundantes na tabela "vehicle" de forma a agilizar o processamento da função. O porquê de ser feito desta forma é explicado de seguida.

Antes desta alteração era necessário verificar dados de diversas tabelas: "vehicle", "notification", "brand_model". Era necessário validar todas as notificações (quatro por cada veículo), verificar a que veículo estavam associadas e validar que plano de manutenção estava associado ao veículo (se o definido pela marca, e que está presente na tabela "brand_model", ou se o plano é personalizado, presente na tabela "vehicle"). Mediante os dados do plano de manutenção seriam feitas as comparações referentes às regras das notificações, de forma a validar quais seriam colocadas como ativas.

Com as várias tabelas com centenas ou milhares de dados, o número de acessos aos discos aumenta, o que poderia trazer alguma lentidão (embora esta função esteja agendada para ser executada durante a noite).

No modelo de dados atual, para evitar este possível constrangimento, existem quatro campos novos na tabela "vehicle": "next_repair_mileage", "next_repair_date", "next_timing_belt_mileage", "next_timing_belt_date", sendo eles redundantes pois é possível saber estes valores com diversos cálculos utilizando os dados do veículo e do plano de manutenção associado. Tal como o nome de cada um indica, estes campos guardam a quilometragem e data da próxima manutenção programada e mudança de correia de distribuição.

Estes campos são apenas atualizados quando o utilizador insere um novo veículo e introduz dados referentes à manutenção anterior do veículo, ou quando a oficina conclui uma reparação e indica que se tratou duma manutenção programada ou mudança da correia de distribuição.

Com esta implementação apenas é necessário verificar se todos os veículos que estão ativos estão de acordo com as regras das notificações, atualizando depois a tabela "notification" com as devidas alterações.

O evento “CheckNotifications” foi implementado com o seguinte código.

```
CREATE EVENT CheckNotifications
ON SCHEDULE
EVERY 1 day
COMMENT 'verifica as notificacoes dos veiculos'
DO
call CheckNotifications;
```

3.4. Aplicação WEB

Nesta secção descreve-se como a aplicação web foi desenvolvida, destacando-se a arquitetura da mesma, os diferentes módulos que constituem esta aplicação e os respetivos componentes. Similarmente à secção "Procedimentos e funções" da base de dados (mas sem ser em forma de tabela), serão também descritas as funções que cada componente possui.

A aplicação web possui quatro módulos, sendo dois deles genéricos: Registo e Login (pois cada uma das entidades: cliente e oficina, pode utilizar), e os restantes dois restritos a cada uma das entidades (cliente e oficina). Além destes quatro módulos, destacam-se também dois componentes genéricos, que são sempre utilizados pelos dois módulos (cliente e oficina): "service" e "user.service".

Alguns dos módulos têm um diagrama para auxiliar na compreensão do funcionamento do mesmo. Nestes diagramas por vezes podem não aparecer os dois componentes genéricos referidos: "service" e "user.service", sendo que o leitor deverá ter em consideração que de facto estes dois módulos são utilizados e estão subentendidos.

A figura 3.3 representa o diagrama de arquitetura genérico da aplicação web, em que o utilizador poderá aceder à sua área de trabalho como cliente ou oficina, mas que primeiramente terá de possuir um registo no sistema, e validar-se sobre o mesmo através do módulo de Login.

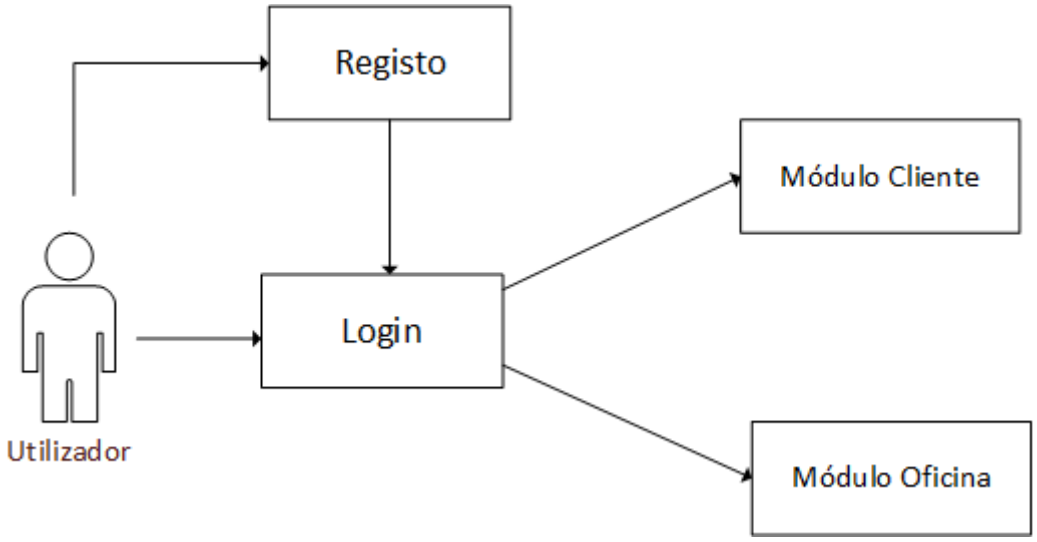


Figura 3.3 – Diagrama arquitetura Aplicação web.

3.4.1. Módulo de Registo

O registo só poderá ser efetuado através da aplicação web. O utilizador deverá fornecer dados válidos, e no ato do registo, é também criado um utilizador para o respetivo acesso. A oficina também pode registar qualquer cliente, sendo que o sistema deverá primeiramente validar se o número de identificação fiscal (NIF) já existe (esta opção apenas está disponível quando é criada uma nova reparação). Neste caso, se o NIF introduzido pela oficina não existir, a aplicação irá criar um novo cliente sem criar qualquer utilizador. Após este registo de cliente, é emitido um código (*token*) para que o cliente possa associar um utilizador a este novo registo de cliente. A oficina tem a possibilidade de visualizar este *token*, bastando pesquisar pelo NIF do cliente. O processo de registo, tal como o nome indica, permite o registo na plataforma de qualquer utilizador das duas entidades, cliente e oficina.

3.4.2. Módulo de Login

O processo de login deverá ser feito com recurso a *username* e *password*. Se o utilizador for válido, é gerado um *token* de sessão para que este possa utilizar os diversos serviços da aplicação. Se o utilizador for válido, mas estiver inativo, é devolvida a mensagem de utilizador inativo. Este processo aplica-se tanto à aplicação web como à aplicação móvel, e é sempre devolvido o identificador do utilizador e o *token* de sessão. O *token* de sessão tem uma validade de 45 minutos para a aplicação web, no segmento dos clientes, e de 12 horas no segmento das oficinas. Para a aplicação móvel este tempo de validade é de 1 ano, pois é necessário que o dispositivo móvel envie periodicamente os dados referentes ao automóvel para a base de dados, sem que o utilizador faça recorrentemente login nesta.

Se o utilizador fizer login novamente na aplicação móvel ou aplicação web, um novo *token* é gerado, e o *token* antigo é colocado como expirado.

Sempre que um login é efetuado com sucesso, o componente genérico – "user.service", guarda os dados de login do utilizador (*token*, ID (identificador) de utilizador, *username*, se está ou não "*logged in"* e se tem perfil de administrador).

3.4.3. Componentes e rotinas genéricos(as)

Existem dois componentes que são utilizados sempre nos dois módulos principais (cliente e oficina) da aplicação web, sendo eles os seguintes:

- "service" – cuja função é invocar os diversos URI disponíveis no *webservice* do servidor web desenvolvido em NODEJS, contendo todas as funções para "consumir" todos os URIs disponíveis. Qualquer módulo da aplicação web possui funções que consomem *webservices* específicos.
- "user.service" – que controla os acessos referentes aos utilizadores, guardando *tokens* de sessão, identificadores de utilizador, entre outros. Guarda também a data atual proveniente do servidor de base de dados no momento do login. Esta data atual serve para que em diversas operações, o sistema saiba qual o ano atual.

A rotina genérica sempre presente nos componentes dos dois módulos, cliente e oficina, representa uma função que executa o fim de sessão do utilizador caso este a termine ou caso esta expire – seja pelo fim do tempo de sessão ou pelo facto do utilizador ter iniciado a sessão num outro dispositivo ou até no mesmo dispositivo.

É essencial que os dois componentes e esta rotina estejam disponíveis para serem utilizados nos diferentes módulos da aplicação web, pois contêm funções que permitem a utilização correta da aplicação por qualquer um dos diferentes utilizadores, garantindo a utilização segura e coerente da aplicação.

Além dos dois módulos referidos acima, que são genéricos a qualquer utilizador, existem outros dois módulos, cada um direcionado para o cliente e oficina, e que possuem as suas próprias funcionalidades e os seus componentes. O funcionamento destes módulos será explicado seguidamente.

3.4.4. Módulo Cliente

Este módulo contém todas as funcionalidades que qualquer cliente consegue realizar sobre a plataforma, sendas elas: visualização dum *dashboard*, visualização de todas as reparações, visualização e criação de veículos e visualização de estatísticas referentes aos seus veículos (quilometragem e consumos de combustível). Qualquer referência posterior ao termo utilizador, refere-se a um utilizador do tipo cliente.

A figura 3.4 representa o diagrama da aplicação web quando se trata dum utilizador do tipo cliente.

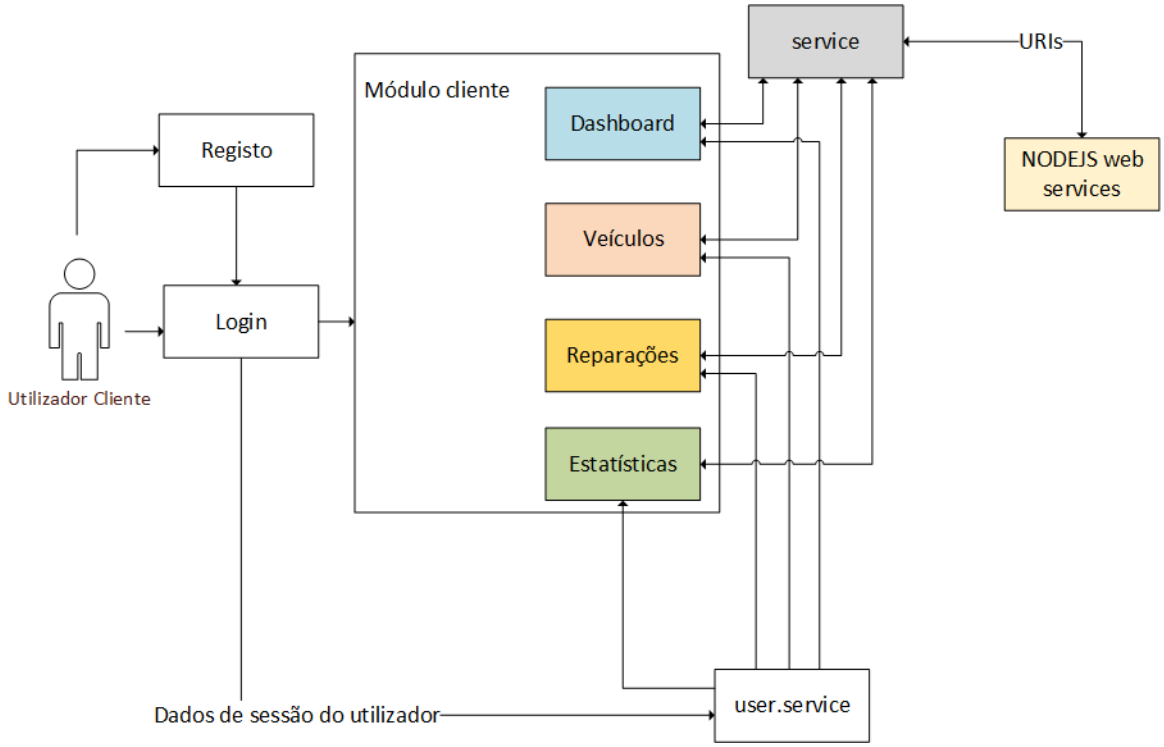


Figura 3.4 – Diagrama de funcionamento da aplicação web para clientes.

3.4.4.1. Componente Dashboard

Neste componente o utilizador poderá visualizar informações referentes aos seus veículos, informações estas que contêm o estado das suas reparações e também avisos referentes a revisões que estejam próximas. No caso do utilizador ser administrador, poderá visualizar estas informações de todos os seus veículos; caso seja utilizador sem direitos de administração, apenas pode consultar estas informações para o veículo associado.

O "Dashboard" (figura 3.5) contém os avisos da proximidade das revisões dos veículos do utilizador, e a lista de reparações dos seus veículos que ainda não estejam concluídas. O utilizador poderá colocar as notificações como "vistas", e só depois poderá apagar estas notificações. As notificações apagadas ficam inativas e como "não vistas".

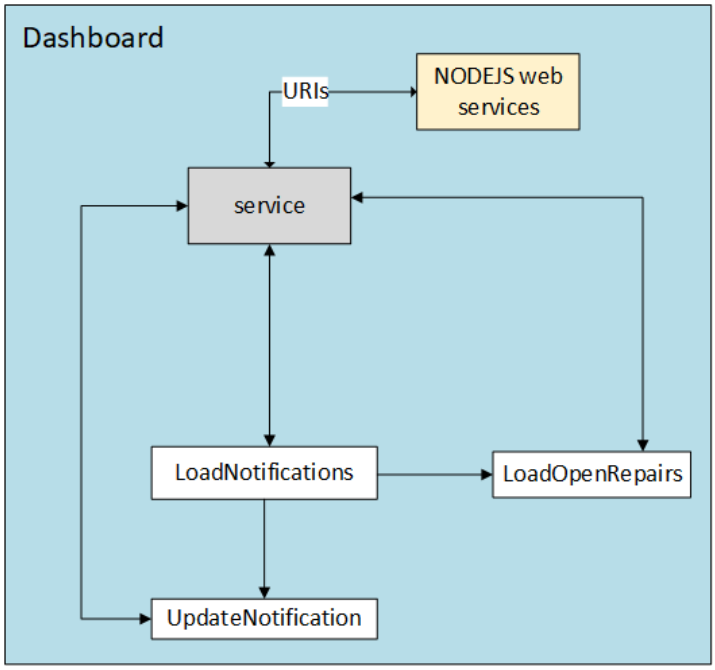


Figura 3.5 – Diagrama de funcionamento do componente "Dashboard".

O cliente quando acede ao seu *dashboard*, é invocada a função "*LoadNotifications"* que irá carregar as notificações ativas referentes aos seus veículos. Seguidamente a esta função, é invocada a "*LoadOpenRepairs"* de forma a carregar todas as reparações dos veículos do cliente que estejam no estado "Registada" ou "Em progresso". A função "*UpdateNotification"* é sempre invocada quando o utilizador atualiza o estado da notificação, seja para colocá-la como "Vista" ou para apagá-la (colocando-a na base de dados como inativa). A alteração do estado da notificação segue a seguinte regra de negócio: só é possível apagar a notificação depois do utilizador a definir como "Vista".

3.4.4.2. Componente Veículos

Este componente, representado na figura 3.6, possui dois subcomponentes: lista de veículos e novo veículo. O diagrama seguinte demonstra as duas opções que o cliente pode escolher, mostrando também as funções mais pertinentes que cada módulo possui.

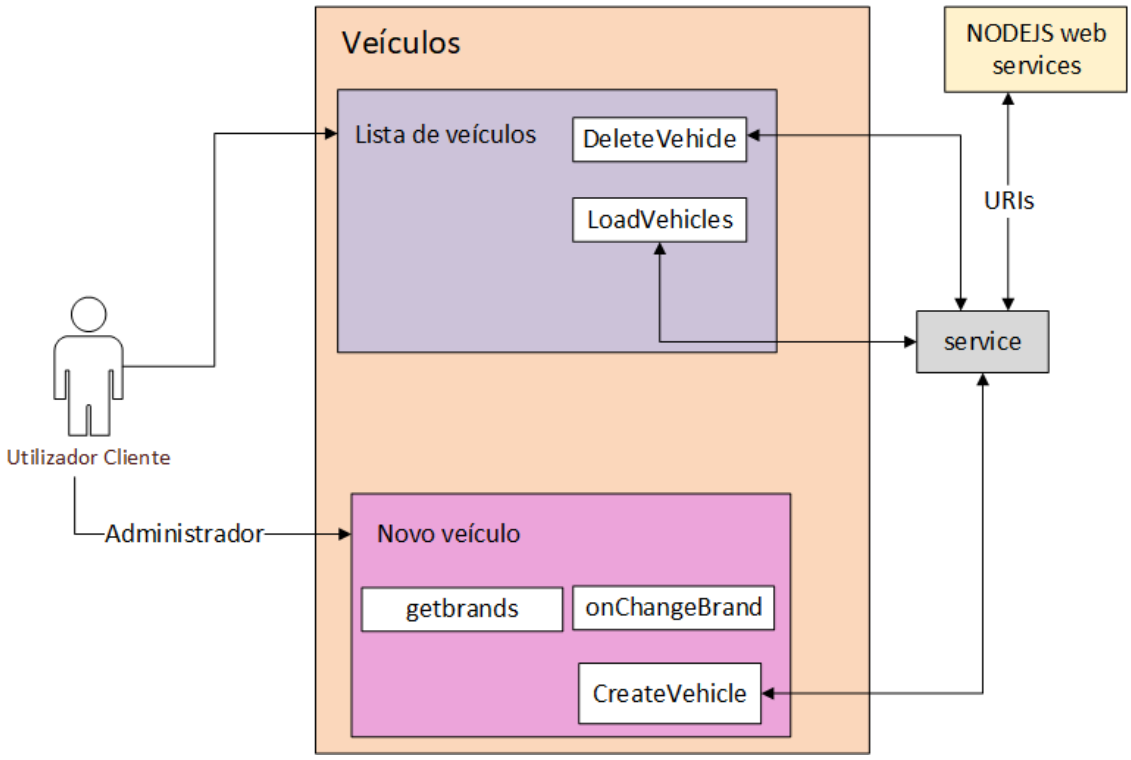


Figura 3.6 – Diagrama de funcionamento do componente "Veículos".

3.4.4.2.1. Subcomponente "Lista de veículos"

Este subcomponente permite a qualquer utilizador visualizar com detalhe todos os seus veículos associados (caso seja administrador, serão todos os veículos, caso não seja administrador, apenas o veículo associado). Além das características de cada veículo, o utilizador consegue também verificar a previsão, seja em data e quilómetros, da próxima manutenção (revisão) e correia de distribuição. Existirá também um painel que indica ao utilizador qual o plano de manutenção associado ao veículo (se o definido pelo fabricante ou personalizado). Este plano de manutenção define o intervalo de revisões em quilómetros e anos de cada veículo.

É invocada a função "*Load Vehicles*", que irá devolver todos os veículos do cliente caso este seja administrador, e apenas o veículo do utilizador caso este não seja administrador.

A função "*DeleteVehicle*" coloca o veículo como inativo, passando este veículo a estar indisponível tanto para o cliente como para a oficina (caso a oficina faça uma pesquisa pela matrícula do veículo por exemplo). Esta função apenas está disponível caso o utilizador seja do tipo administrador.

3.4.4.2.2. Subcomponente "Novo veículo"

Este subcomponente está apenas disponível para utilizadores com o perfil de administrador e permite registar veículos na plataforma, visto que poderá haver utilizadores que não pretendam a utilização da aplicação móvel e sistema embebido e apenas necessitem de algo que forneça um histórico de reparações dos seus veículos e um auxiliar na gestão da manutenção dos mesmos. Também, só um utilizador com estas permissões é que poderá apagar qualquer veículo (sendo que no modelo de dados o veículo não será eliminado, mas sim colocado como inativo). Cada vez que é registado um novo veículo, são criadas quatro notificações padrão. Estas notificações são definidas como inativas e como "não lidas". Cada notificação será colocada como ativa quando o evento "*check_notifications*" as definir como tal, e será colocada como lida quando o utilizador a verificar e indicar ao sistema que tomou conhecimento desta mesma notificação (já explicado no componente "*Dashboard*").

Neste subcomponente o utilizador deverá escolher duma lista de marcas (lista essa que é devolvida pela função "*getbrands*") a marca do veículo, e de seguida o modelo

do mesmo presente numa outra lista, lista essa que é preenchida pela função "*onChangeBrand*", que apenas contém todos os modelos da marca que o utilizador escolheu inicialmente. O utilizador deverá preencher os restantes dados do veículo, como a quilometragem, o ano e mês, entre outros. Poderá também personalizar a quilometragem de revisão do seu veículo ou caso não pretenda esta personalização, o plano de manutenção associado é definido pela marca e modelo escolhidos anteriormente.

3.4.4.3. Componente Reparações

Este componente, tal como o nome indica, possibilita a visualização das reparações (e seus detalhes) dos veículos do cliente. De forma a garantir que não haja problemas de performance quando a base de dados devolve estas reparações, o utilizador tem de inserir a matrícula do veículo e pode escolher o número de resultados que pretende visualizar (25 é o valor por omissão, e 100 o valor máximo). A figura 3.7 representa este campo de pesquisa mencionado anteriormente.

Figura 3.7 – Campo de pesquisa e máximo de resultados.

3.4.4.4. Componente Estatísticas

O componente "Estatísticas" pretende mostrar em forma de gráfico cada uma das métricas do veículo que são recolhidas pelo sistema embebido. Neste componente, o utilizador escolhe o veículo e a métrica, e a aplicação web invoca a respetiva função, de forma a devolver os dados que o utilizador solicitou. Visto que no momento da escrita deste relatório não existiam dados reais, foram gerados dados aleatórios como representação duma viagem real. Assim a figura 3.8 representa os valores da velocidade (eixo YY) em função do tempo (eixo XX – intervalos de 1 minuto), demonstrando o layout final deste componente:

Figura 3.8 – Gráfico "Velocidade" componente "Estatísticas".

3.4.5. Módulo Oficina

Este módulo serve de acesso a qualquer oficina, dispondo aos seus utilizadores as funcionalidades de: registar reparações, criar clientes, alterar o estado das reparações e criar veículos. Este módulo deverá conter todas as funcionalidades de auxílio à gestão das reparações numa oficina. Qualquer referência posterior ao termo utilizador, refere-se a um utilizador de oficina.

O utilizador, após o seu *login*, terá um *dashboard* com a informação acerca das reparações que ainda não estão concluídas. Ao clicar sobre cada uma destas, são mostrados detalhes acerca da reparação.

Poderão existir clientes que não têm ainda registo na plataforma web, e por isso, tal como descrito no processo de registo, qualquer oficina pode registar um novo cliente. Além de clientes, é possível depois registar um novo veículo que possa ser associado a este cliente e por conseguinte a uma nova reparação.

A oficina poderá registar novos veículos e novos clientes, mas não poderá desassociar os clientes dos veículos, exceto aquando do registo duma nova reparação se o cliente final for diferente do que está já associado ao veículo, se por engano ao registar este novo cliente introduzir os dados erradamente, poderá apagar esta última inserção do cliente. A figura 3.9 representa o módulo oficina com as respectivas funcionalidades.

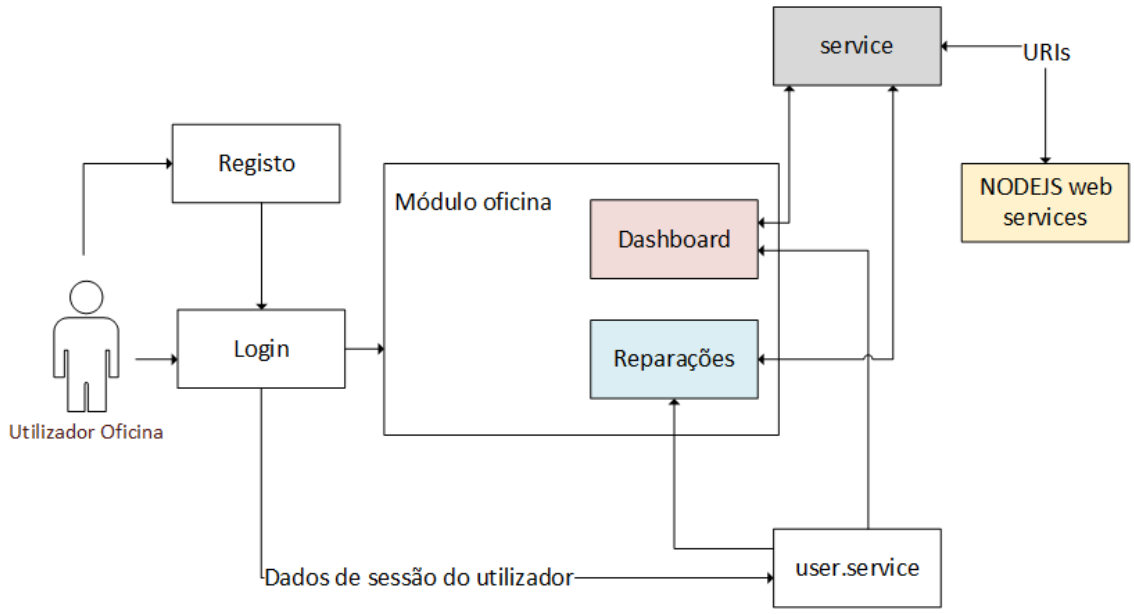


Figura 3.9 – Diagrama de funcionamento da aplicação web para oficinas.

3.4.5.1. Componente Dashboard

Este componente (representado na figura 3.10) deverá mostrar ao utilizador as reparações que ainda estão ativas (registadas e em progresso), sob forma de lista. Ao clicar num elemento da lista de reparações, poderá visualizar outros detalhes da reparação e também alterar o estado desta (Em progresso, Concluída ou Cancelada). O estado "Registada" apenas é possível definir quando uma nova reparação é registada.

É neste componente que depois o utilizador pode finalizar as suas reparações, tendo também um elemento de pesquisa, capaz de filtrar os veículos ou por matrícula ou marca.

Existem várias regras de negócio referentes à alteração do estado das reparações, tal como o facto das reparações depois de serem iniciadas (Em progresso) apenas puderem ser canceladas ou concluídas, e as reparações que estejam registadas apenas poderem ser iniciadas ou canceladas. Estas alterações provocam atualizações nos registos das reparações presentes na base de dados, com o intuito de fornecer feedback aos utilizadores dos tempos das reparações e horas de início e conclusão.

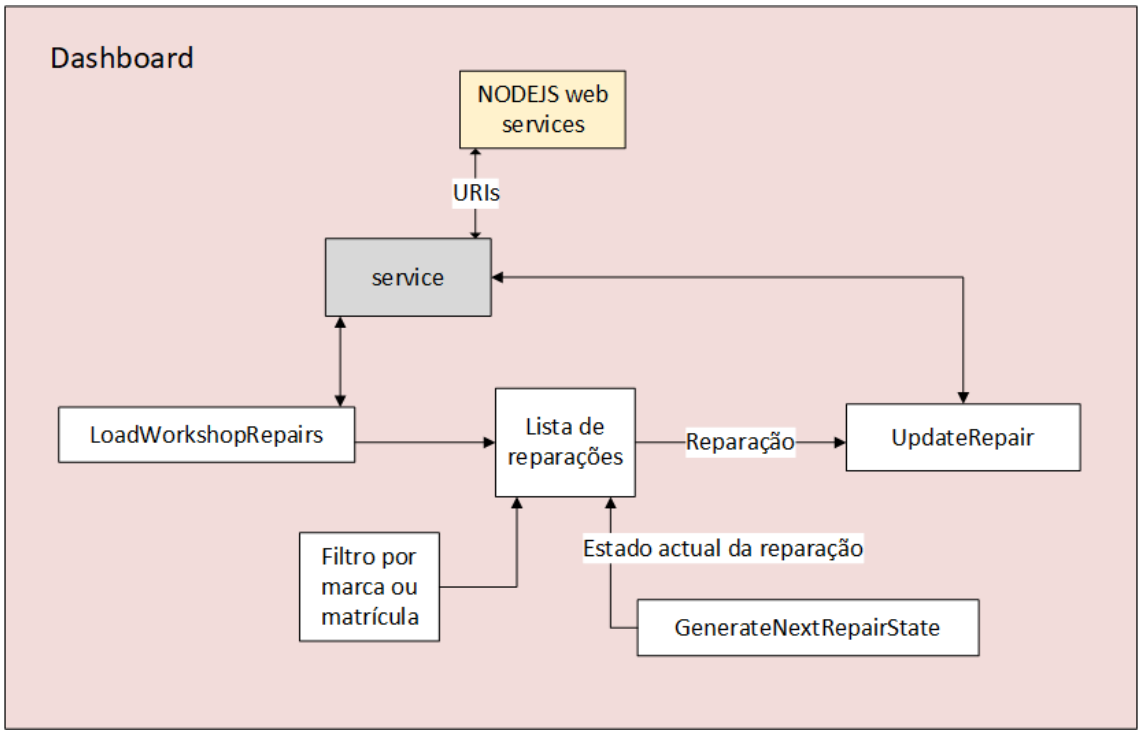


Figura 3.10 – Diagrama de funcionamento do componente "Dashboard – Oficina".

A função "*LoadWorkshopRepairs"* carrega uma lista de todas as reparações da oficina que não estejam nem concluídas nem canceladas. De forma a facilitar o uso desta lista, o utilizador pode filtrar por marca ou matrícula a lista de reparações. A função "*GenerateNextRepair"*, gera todos os seguintes estados possíveis que cada reparação poderá ter, tendo como base o estado atual em que a reparação está. Quando o utilizador altera o estado duma reparação, a função "*UpdateRepair"* encarrega-se de comunicar com o módulo "service" para utilizar o URI destinado à atualização do estado da reparação na base de dados.

O ciclo de vida duma reparação começa como "Registada", de seguida é colocada "Em progresso" e por último como "Concluída", mas a qualquer momento (exceto quando está já concluída) o utilizador pode cancelar a reparação. A figura 3.11 representa esse ciclo de vida.

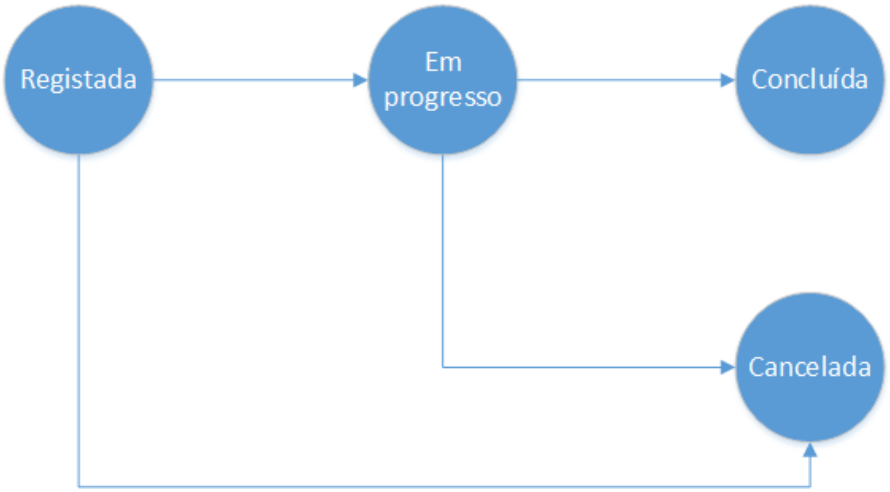


Figura 3.11 – Ciclo de vida duma reparação.

Para o utilizador registar qualquer reparação como "Concluída" terá de preencher pelo menos o campo correspondente ao total da reparação, que representa o custo desta em euros.

Quando uma reparação é dada como concluída, na tabela veículo poderão ser atualizados diversos campos: quilometragem e data da última revisão e quilometragem e data da última correia de distribuição, para que possam surgir as futuras notificações referentes às tarefas de manutenção do veículo. Estes campos apenas são atualizados caso

o utilizador selecione os campos "Manutenção programada" ou "Mudança da distribuição" na reparação que está a definir como concluída.

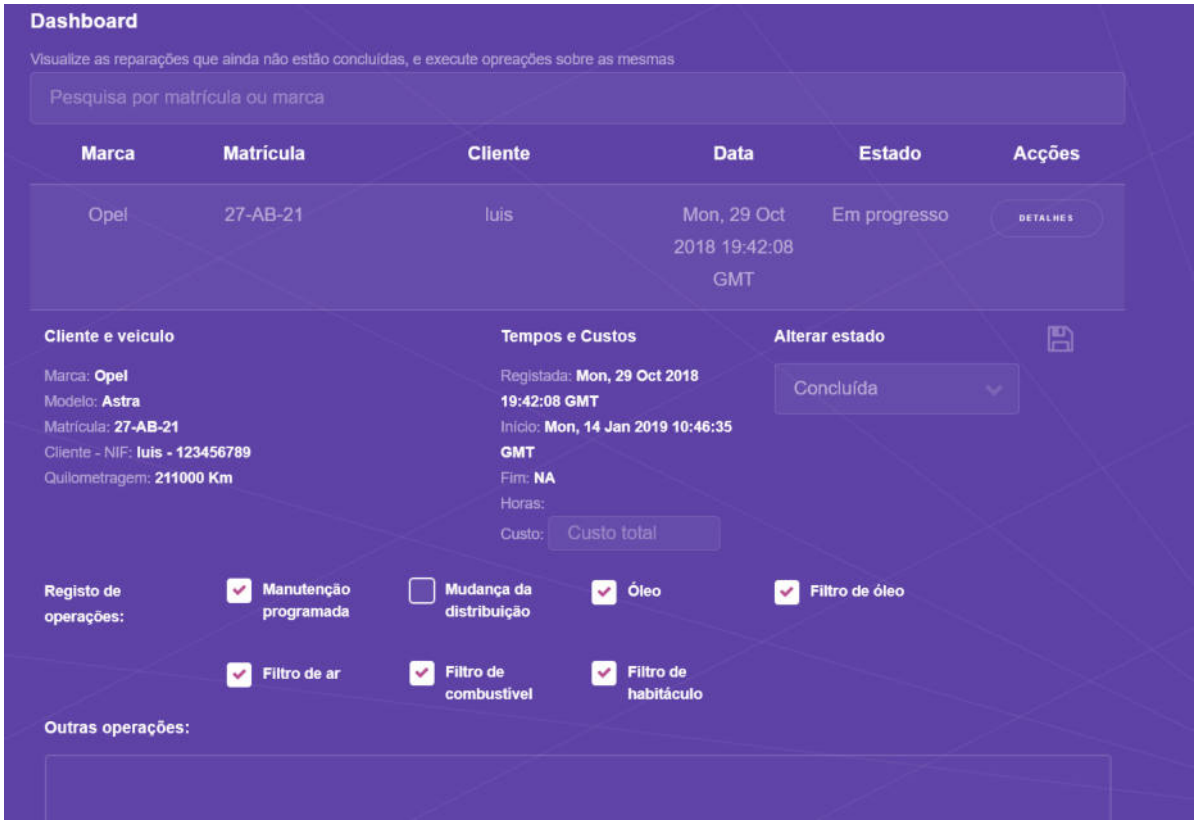


Figura 3.12 – Protótipo "Dashboard" do módulo Oficina.

3.4.5.2. Componente Reparações

Este componente (representado na figura 3.13) contém dois subcomponentes: lista de reparações e nova reparação. Tal como os nomes indicam, no primeiro ("lista de reparações") o utilizador conseguirá consultar todas as reparações efetuadas na sua oficina de forma detalhada; no segundo ("novas reparações"), o utilizador poderá registar uma nova reparação, sendo que neste também consegue registar um novo cliente e um novo veículo. Estas opções são essenciais pois poderão existir clientes da oficina que ainda não possuem qualquer registo na plataforma web, e pelo modelo de dados existe a obrigatoriedade de existir um cliente e um veículo para que qualquer nova reparação seja registada.

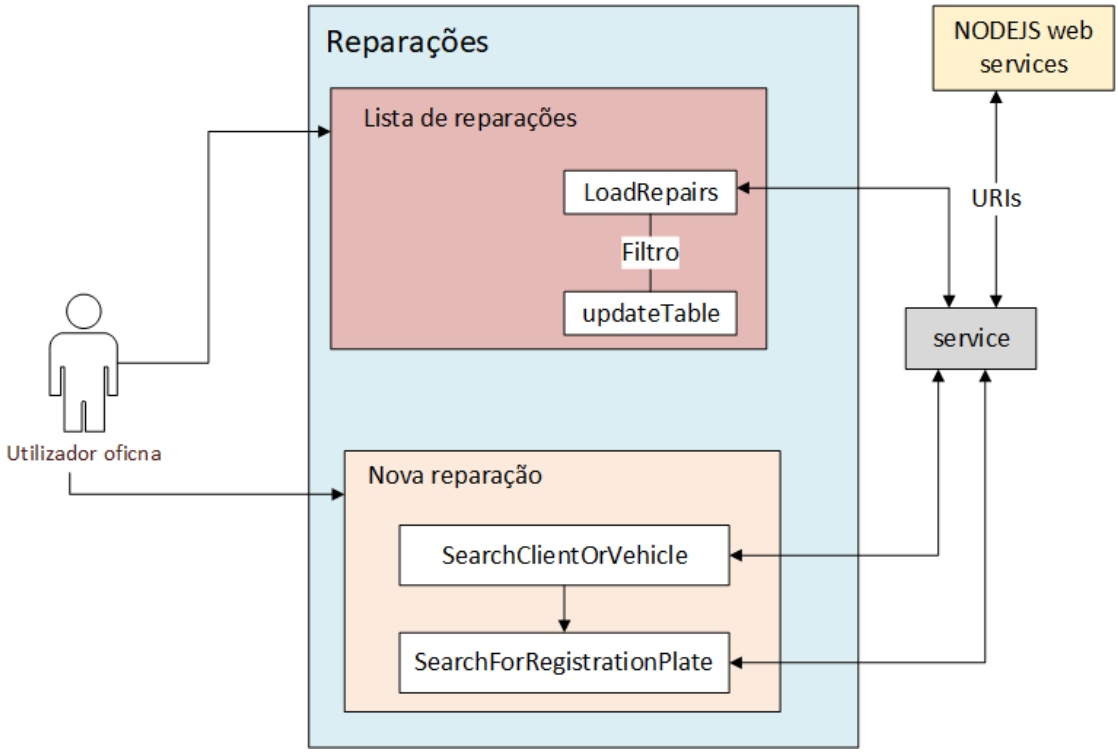


Figura 3.13 – Diagrama de funcionamento do componente "Reparações".

3.4.5.2.1. Subcomponente "Lista de reparações"

Na lista de reparações o utilizador pode verificar as reparações efetuadas na sua oficina. Para isso deverá pesquisar pelo NIF do cliente ou pela matrícula do veículo, sendo que os resultados devolvidos estão limitados a 1000 por uma questão de performance e segurança. Este subcomponente é simples, contendo apenas duas funções:

- *"LoadRepairs"* – carrega da base de dados a lista de reparações afetas ao parâmetro que o utilizador pesquisou.
- *"updateTable"* – filtra a tabela de reparações devolvida acima, pelos campos: "nome do cliente" ou "matrícula do veículo".

3.4.5.2.2. Subcomponente "Nova reparação"

No subcomponente "Nova reparação", o utilizador poderá pesquisar pelo NIF do cliente ou pela matrícula do veículo do cliente, tendo depois a possibilidade de criar novos veículos e novos clientes, e finalizando com o registo de uma nova reparação.

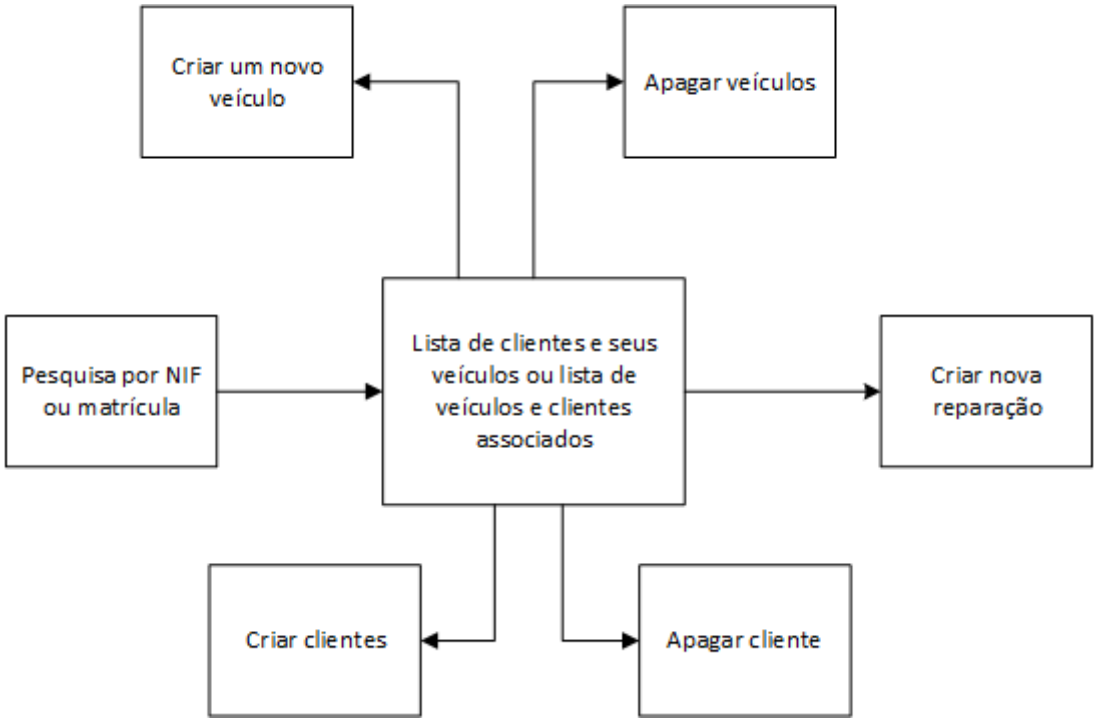


Figura 3.14 – Diagrama genérico de ações do subcomponente "Reparações – Nova Reparação".

Primeiramente este componente começa por invocar o URI que pesquisa pelo NIF do cliente ("*SearchClientOrVehicle"*), e caso não exista qualquer cliente, invoca um segundo URI, que pesquisa pela matrícula do veículo do utilizador ("*SearchForRegistrationPlate"*). Poderia existir a opção do utilizador escolher que campo necessita de pesquisar (se NIF ou matrícula), mas por uma questão de comodismo do utilizador, optou-se por esta forma colocando-se um campo de pesquisa que facilmente o utilizador compreende que dados pode procurar.

Figura 3.15 – Campo pesquisa nova reparação.

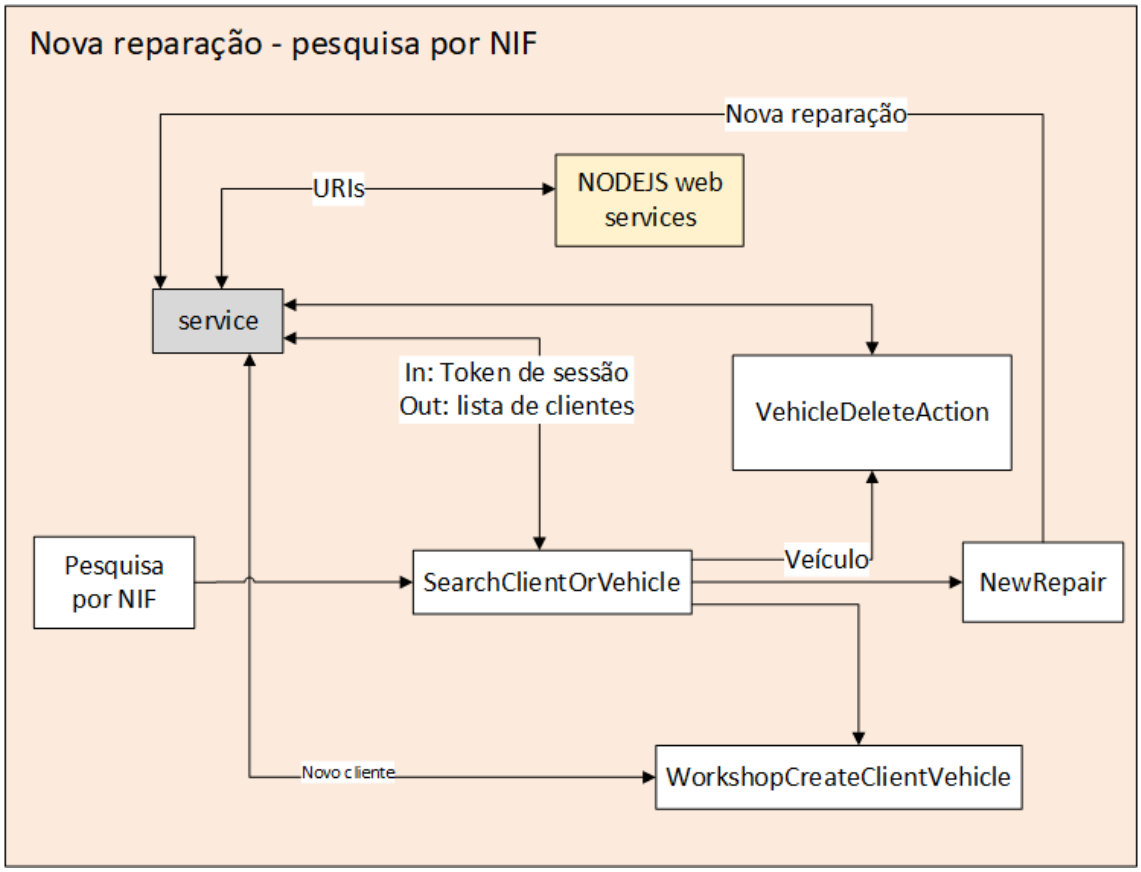


Figura 3.16 – Diagrama de funcionamento nova reparação – pesquisa por NIF.

Se o NIF existir (cujo fluxo de funcionamento é representado pela figura 3.16), serão mostrados os veículos do cliente e o utilizador poderá selecionar um dos veículos que o cliente possua e abrir uma nova reparação de forma que fique associada aos respetivos cliente e veículo ("*NewRepair*"). Caso não sejam devolvidos nenhuns veículos (pois o cliente pode apenas ter-se registado no sistema e não registado nenhum veículo), o utilizador tem também a possibilidade de criar um novo veículo, inserindo apenas a marca, modelo, ano, mês, matrícula e quilometragem atual do mesmo ("*WorkshopCreateClientVehicle*"). Em consequência, o cliente consegue visualizar este novo veículo registado na sua área pessoal da aplicação web. Também, se nesta pesquisa forem devolvidos veículos, o utilizador pode registar uma nova reparação e corrigir a quilometragem do veículo, pois a mesma pode não estar correta ou atualizada, mantendo assim coerência dos dados da reparação com a realidade.

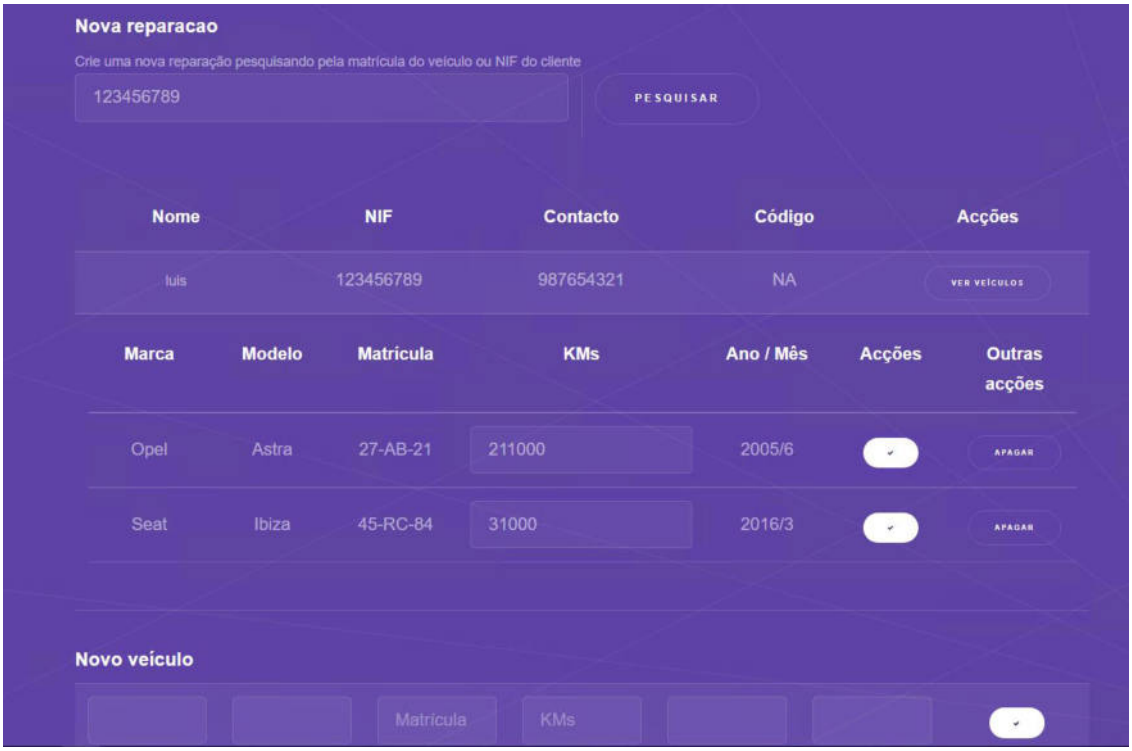


Figura 3.17 – Nova reparação – pesquisa por NIF.

A opção de pesquisar pela matrícula, deve-se também ao facto do veículo poder existir, mas a faturação poder ser lançada com um número de contribuinte diferente. Este fluxo de funcionamento quando é pesquisado uma matrícula é representado pela figura 3.18.

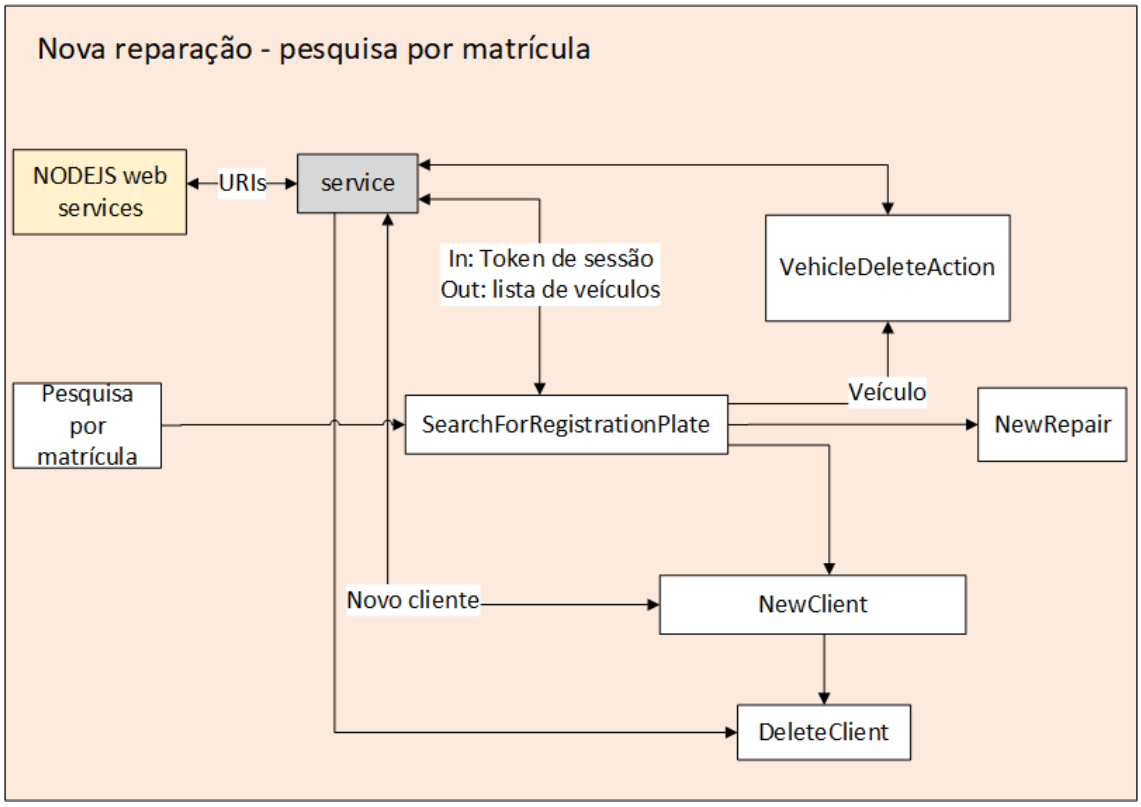


Figura 3.18 – Diagrama de funcionamento nova reparação – pesquisa por matrícula.

Neste caso, a oficina pode registar um novo cliente, inserindo o NIF, nome e contacto, sendo também gerado e mostrado um código (*token*) para que o cliente se possa registar no sistema ("*NewClient"*). Caso no passo anterior a oficina insira erradamente os dados deste cliente, tem também a opção de eliminar este cliente inserido ("*DeleteClient"*).

Poderá ocorrer casos em que poderão aparecer mais que um veículo com a mesma matrícula, e neste caso, além dos dados de cada veículo são também enviados dados referentes aos clientes associados a cada veículo, com a finalidade do utilizador conseguir perceber que veículo se trata.

A oficina tem a autonomia para desativar os registos de qualquer veículo duplicado cujos dados estejam já inconsistentes (uma situação em que o carro foi vendido e mudou de proprietário por exemplo). Os veículos ficam como inativos da pesquisa por parte das oficinas, mas continuam ativos para que os utilizadores possam continuar a ver os dados do seu veículo, impedindo assim que sejam registadas novas reparações em veículos que sejam inválidos. O veículo apenas fica como inativo quando o utilizador (cliente) o eliminar. Cada veículo que seja registado, gera um registo novo na tabela de "vehicle", e por isso haverá veículos diferentes com a mesma matrícula, e cada um terá as suas próprias reparações registadas. Como referido anteriormente, fica a cargo da oficina inutilizar estes veículos inválidos, para que não haja erros no registo de futuras reparações. A função "*VehicleDeleteAction*" tem o objetivo referido anteriormente, sendo genérico a estas duas formas distintas do registo de novas reparações.

No registo de qualquer nova reparação, é possível alterar a quilometragem do veículo que o utilizador escolheu, pois, este dado pode não se encontrar atualizado no momento do registo desta nova reparação.

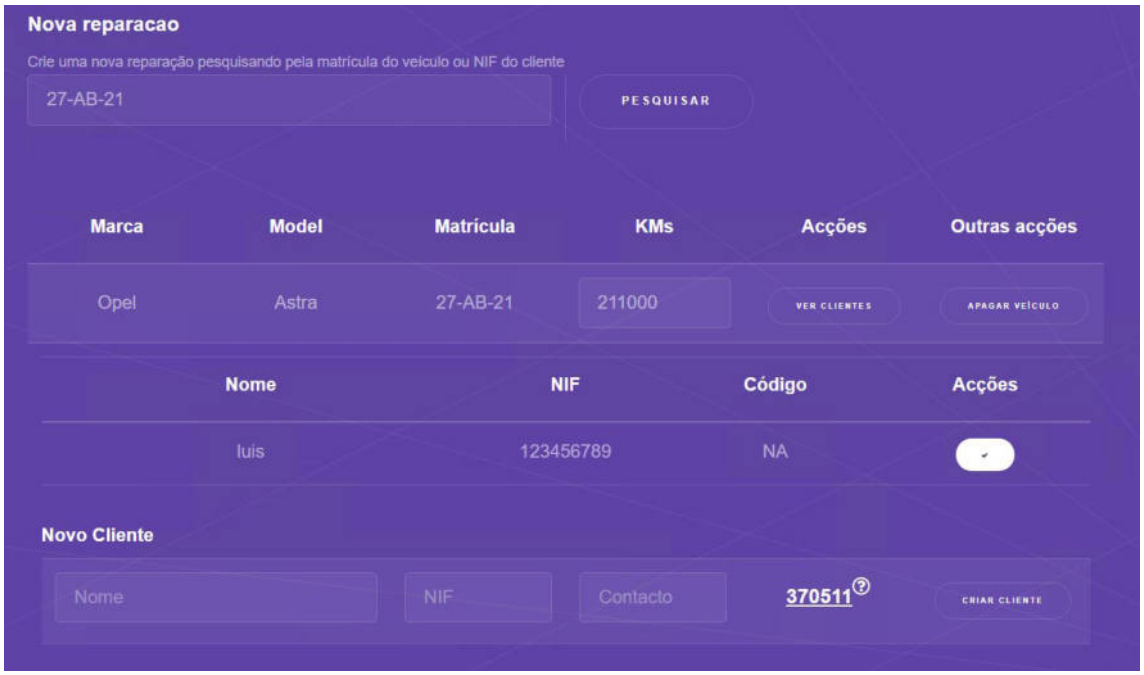


Figura 3.19 – Nova reparação – Pesquisa por matrícula de veículo.

Subentende-se que o componente “user.service” está presente sempre antes de ser invocado qualquer URI, isto porque é sempre necessário o envio do *token* de sessão de forma a validar a autenticidade e validade da sessão do utilizador.

3.5. Aplicação móvel

Aplicação móvel tem como finalidade configurar o veículo do utilizador e também o sistema embebido. Tem um serviço em *background* de forma a receber os dados do sistema embebido (dados estes relativos ao veículo). Caso o Bluetooth não esteja ativo ou fora do alcance, nenhuma ligação pode ser feita até ao sistema embebido, e por isso, a aplicação deverá validar quando o sistema embebido se encontra nas proximidades para que uma conexão seja estabelecida. Isto será feito com tentativas de ligação ao sistema embebido, de forma faseada para economizar bateria no dispositivo móvel. A aplicação móvel faz uso das técnicas de *Threading* para uma melhor usabilidade e toda a comunicação entre o sistema embebido e aplicação móvel é cifrada utilizando uma chave simétrica que é gerada quando um novo utilizador se regista (Stack overflow, 2011).

3.5.1. Arquitetura da aplicação móvel

O diagrama de blocos na figura 3.20 representa os componentes da aplicação móvel com os devidos fluxos de comunicação.

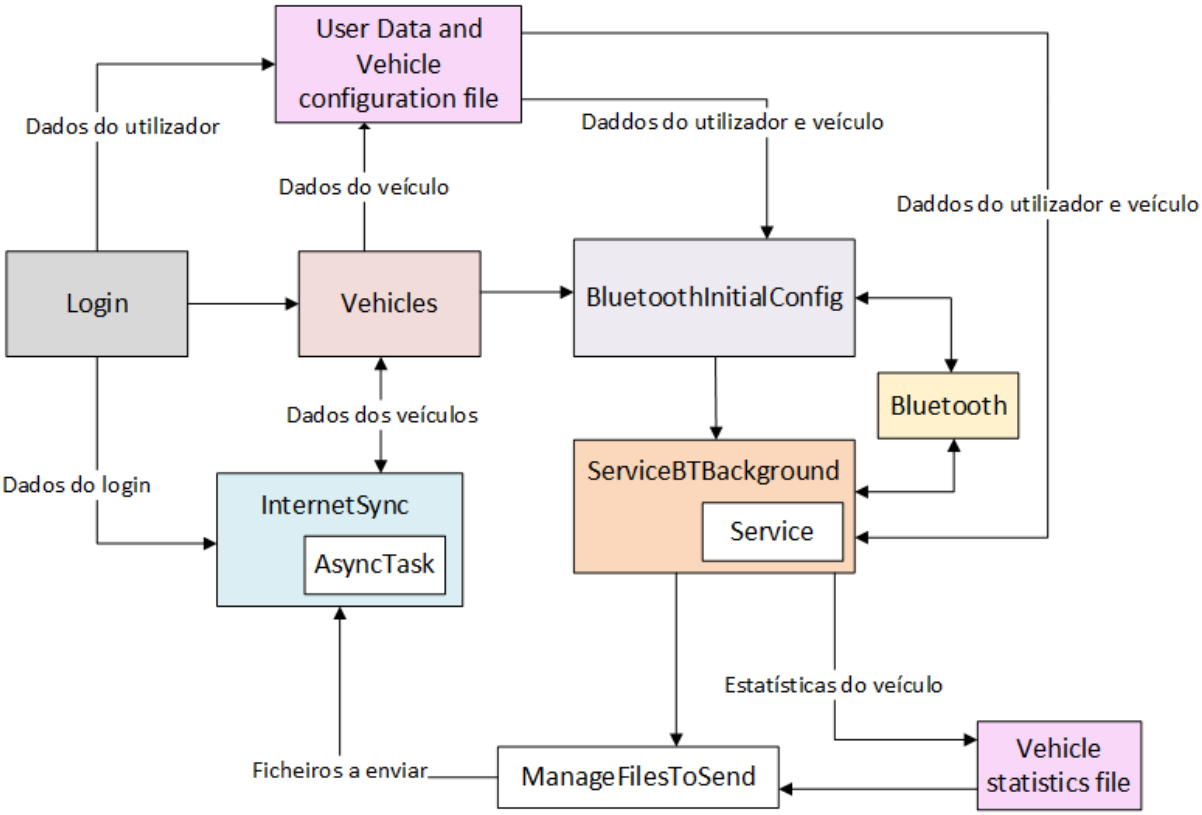


Figura 3.20 – Diagrama de funcionamento da aplicação móvel.

Neste diagrama é possível visualizar os diferentes módulos que constituem a aplicação móvel, e também dois ficheiros essenciais para o seu funcionamento:

- "*User Data and Vehicle configuration file*": este ficheiro contém toda a informação referente ao utilizador (informação esta registada quando se efetua o login), do veículo em causa e também o MAC *Address* do sistema embebido.
- "*Vehicle statistics file*": constitui vários ficheiros, referentes a todas as estatísticas recolhidas pelo sistema embebido.

3.5.2. Módulo InternetSync

Este módulo contém funções da *AsyncTask* e tem como objetivo gerir toda a comunicação entre o dispositivo móvel e o web server desenvolvido em NODEJS, que é realizada sobre a Internet. A invocação dos diversos URIs é feita com recurso à utilização de funções presentes neste módulo e cada módulo que necessite de utilizar estas funções necessita de definir vários parâmetros, sendo eles:

- URI: define o URI a invocar no *webservice*
- Método: define o método – GET, POST, PUT ou DELETE
- Ação: define o módulo origem que invoca estas funções
- Dados: representa os dados a serem transmitidos na comunicação com o *webservice*

É essencial a definição destes parâmetros para uma correta utilização dos diferentes URIs definidos no *webservice*.

3.5.3. Módulo Login

O utilizador através do módulo "Login" autentica-se e acede ao sistema caso este seja válido, sendo guardado posteriormente o ID do utilizador, *token* de sessão e a chave de cifra dos dados que serão transmitidos entre o sistema embebido e a aplicação móvel. Esta ação necessita de uma ligação à internet para conseguir receber estes dados de sessão, sendo estes guardados de seguida num ficheiro, de modo a serem enviados para o sistema embebido. Para gerir esta comunicação através da internet, são invocadas funções do módulo "InternetSync".

O *token* de sessão permite ao utilizador navegar pela aplicação móvel e garantir que este utilizador pode utilizar as funcionalidades presentes no sistema. Este *token*, como já foi referido tem a validade de 1 ano, pois apenas é suposto que o utilizador utilize a aplicação móvel para registar os veículos e configurá-los no sistema embebido.

O ID do utilizador, como o próprio nome indica, é o identificador do utilizador na base de dados. Este identificador é enviado para o sistema embebido, para que este apenas autorize novas configurações ou *Reset* do utilizador válido.

A chave de cifra de dados, será utilizada para cifrar toda a comunicação entre o sistema embebido e a aplicação móvel. Só após esta chave ser enviada para o sistema embebido é que todos os dados futuramente transmitidos serão cifrados.

É recebido também a data atual, de forma a manter coerentes certas funcionalidades presentes na aplicação móvel. Um exemplo é o facto de quando o utilizador registar um veículo, e ao escolher o ano do veículo, o ano máximo é definido pelo ano da data atual recebida.

3.5.4. Módulo Vehicles

Neste, o utilizador pode visualizar os seus veículos registados, bem como adicionar novos veículos. Cada cliente pode possuir vários utilizadores, e quando o acesso é feito através da aplicação móvel, à semelhança do acesso via aplicação web, apenas são devolvidos os veículos do utilizador. Isto deve-se ao facto de na aplicação móvel só se poder configurar o veículo onde o sistema embebido está ligado, evitando assim más configurações de veículos por parte de outros utilizadores pertencentes ao mesmo cliente. Este módulo utiliza funções do módulo "InternetSync", para o registo de novos veículos, e para receber a lista de todos os veículos do utilizador.

Só quando a aplicação móvel receber a lista de veículos do utilizador, é que o próprio utilizador pode selecionar o veículo que deseja configurar no sistema embebido. Também, o utilizador após registar o seu veículo poderá selecioná-lo para que o mesmo seja configurado no sistema embebido.

O seguinte excerto de código mostra a criação dum objeto JSON que contém todos os parâmetros do veículo que o utilizador escolheu para configurar no sistema embebido:

```
vehicle_data.put("id_vehicle", vehicle.getString("id"));
vehicle_data.put("mileage", vehicle.getInt("mileage"));
vehicle_data.put("displacement",
vehicle.getInt("displacement"));
vehicle_data.put("fuel", vehicle.getString("fuel"));
```

3.5.5. Módulo Bluetooth

Neste módulo o utilizador conseguirá validar que o sistema embebido já está emparelhado com o seu dispositivo móvel, e poderá selecionar para que sistema embebido remoto é que serão enviados os seus dados de sessão e do respetivo veículo, sendo que esta opção apenas está disponível quando o utilizador, anteriormente no módulo "Vehicles", seleciona o veículo que deseja configurar.

Estão presentes as funções necessárias para o início de uma ligação Bluetooth, sendo de seguida invocadas as funções do módulo "BluetoothInitialConfig".

3.5.6. Módulo BluetoothInitalConfig

Quando é iniciada a primeira comunicação Bluetooth entre o sistema embebido e a aplicação móvel são enviados os dados do utilizador e do veículo, para que o primeiro se configure. Invocando também funções do módulo *Bluetooth* para que esta comunicação via Bluetooth seja possível, este módulo inicia de seguida o serviço representado pelo módulo "ServiceBTBackgroud".

É gerado um pacote JSON que contém informação referente ao utilizador e ao veículo, informação esta, que irá permitir ao sistema embebido validar o utilizador quando a aplicação móvel se liga novamente a este e também manter a coerência de que veículo o sistema embebido está a recolher as métricas. O excerto de código abaixo, demonstra a definição desse pacote, que será enviado para o sistema embebido:

```
final_configs.put("id",1);
aux.put("user_data",user_data.getUserData());
aux.put("vehicle_data",user_data.getCarConfigs());
aux.put("connection",user_data.getConnection());7
final_configs.put("data",aux);
```

Estas informações são essenciais para o auxílio da configuração inicial do sistema embebido. Uma informação a destacar neste pacote JSON é a chave simétrica (que está guardada na base de dados e enviada para a aplicação móvel aquando do login do utilizador) cuja função é a cifra/decifra dos dados estatísticos do veículo que são transmitidos entre a aplicação móvel e o sistema embebido (já referido no módulo *Login*).

3.5.7. Módulo ServiceBTBackground

Este módulo é responsável por gerir toda a comunicação Bluetooth entre a aplicação móvel e o sistema embebido, depois da primeira ligação ser efetuada pelo módulo "BluetoothInitialConfig". Este, ao invocar funções presentes no módulo *Bluetooth* para que a comunicação seja iniciada, invoca de seguida as suas próprias funções para a gestão desta comunicação. Este módulo, corresponde a um serviço no Android e ficará sempre "activo", gerindo sempre a ligação e garantindo a poupança de energia no caso de quebra desta mesma ligação. Este modo "ativo" significa que o serviço em intervalos de 2 minutos, tenta iniciar a ligação ao sistema embebido.

Ao utilizar o elemento Android: "*Notification*", fornece feedback ao utilizador sobre o estado da sua ligação ao sistema embebido. Embora seja um serviço, e o mesmo deva ficar sempre ativo, acontece que este serviço pode ser terminado assim que a aplicação que o invoca termine. Por isso é necessário incluir a *flag* "*START_STICKY*", que indica ao sistema operativo Android que deve arrancar este serviço mais tarde depois deste ser fechado. Cada vez que a ligação ao sistema embebido é perdida, este módulo é responsável por aguardar que o sistema embebido esteja novamente disponível, e assim que o esteja, deverá ligar-se a este. Cada vez que uma nova ligação é iniciada, são sempre enviados os dados do utilizador, de modo que o sistema embebido valide a autenticidade deste utilizador, pois torna-se necessário que mais nenhum utilizador, exceto o que primeiramente configurou o sistema embebido possa aceder as informações recolhidas por este.

Esta forma de segurança permite que o utilizador mesmo que troque de dispositivo móvel se possa ligar novamente ao sistema embebido e configurá-lo; e também que mais nenhum utilizador se ligue ao sistema embebido e altere qualquer uma das suas configurações.

Assim que o serviço receba as informações provenientes do sistema embebido, guarda-as em vários ficheiros e invoca o módulo "ManageFilesToSend" para que através do módulo "InternetSync", envie os dados destes ficheiros para a base de dados no *backend*.

O módulo "ManageFilesToSend" corresponde a uma classe com extensão de *Thread*, que além de tratar do envio das estatísticas para a base de dados, valida primeiramente e de forma constante que ficheiros existem para ser enviados, e caso haja

uma falha na ligação à Internet, ou não existam ficheiros para serem envidos, aguarda 5 minutos até executar as funções referidas anteriormente.

3.6. Sistema Embebido

Como já foi referido, este componente do projeto será composto por um sistema embebido – Raspberry Pi com um controlador CAN e um RTC (*real time clock*). O controlador CAN tem como função gerar mensagens OBD encapsuladas no protocolo de *Layer* 2 CAN, enviá-las para o veículo e receber a resposta a estas mensagens. Como o Raspberry Pi não possui um relógio interno, um RTC externo será utilizado para este efeito. Cada mensagem gerada, deverá aguardar pela sua resposta até gerar uma nova mensagem. Como se trata de uma interface de diagnóstico e mesmo com a capacidade até 500 kilobits/s da rede CAN, não é pretendido o congestionamento desta por motivos de segurança automóvel. O sistema embebido deverá guardar informação dos clientes Bluetooth que estão ativamente ligados a si, devendo apenas enviar a informação para o cliente correto. A biblioteca Python "*PyBluez*" responsável pelo controlo do Bluetooth no Raspberry Pi consegue perceber quando um cliente perde a ligação ao sistema embebido. Assim, mesmo que o cliente Bluetooth não esteja ativamente ligado, o sistema embebido não deverá parar de recolher estatísticas do veículo em que está ligado. O sistema embebido deverá guardar as estatísticas do veículo de minuto a minuto.

Em termos de software, o desenvolvimento do sistema embebido foi abordado faseadamente, pretendendo que o mesmo atingisse as funcionalidades básicas necessárias para o utilizador. As diferentes fases são as seguintes:

1. Autoconfiguração e *reset*: o sistema embebido estará sempre à escuta de pedidos de *reset* e também de novas autoconfigurações dos veículos. Este *reset* só pode ser feito pelo utilizador que configurou primeiramente o sistema embebido, visto que o acesso a este só é feito após login na aplicação móvel. Apenas um veículo pode ser configurado simultaneamente no sistema embebido, e por isso um novo veículo fará com que qualquer configuração anterior neste seja perdida.
2. Deteção do estado do veículo: o sistema embebido deverá detetar que o veículo já se encontra ligado e com o motor a trabalhar.
    a. O sistema embebido aguarda que a rotação do motor seja maior que 0 ao enviar *queries* OBD2 com o código das rotações por minuto e validando a resposta recebida, em intervalos de 1 segundo. Assim que o motor do veículo é ligado, o sistema embebido irá executar um procedimento com

o objetivo de detetar que sensores o veículo possui, para depois se poder autoconfigurar.

3. Módulo de deteção dos sensores: o sistema embebido deverá efetuar diversas *queries* OBDII de forma a perceber que sensores o veículo possui. A deteção destes sensores é importante para que o sistema embebido apenas transmita as mensagens necessárias para o cálculo do consumo de combustível e quilometragem.
4. Cálculo da distância percorrida: tem como finalidade calcular a distância percorrida pelo veículo através do sensor de velocidade deste.
    a. Assim que o Pi receber um valor superior a 0 do sensor da rotação do motor, enviará uma mensagem de modo que a resposta seja o valor da velocidade atual do veículo. Esta *frame* OBDII tem o PID = 0D.
    b. O sistema embebido soma todos os valores de velocidade que receber durante 5 segundos e depois divide pelo total de pacotes recebidos para fazer uma média da velocidade. A distância é calculada multiplicando este valor médio por 5 segundos e dividindo por 3600. Para a distância total, é feito o somatório destas distâncias percorridas durante os 5 segundos. A cada 5 segundos, o veículo envia também uma *frame* com o PID das rotações por minuto para perceber se o veículo ainda está a trabalhar. Assim, no pior dos cenários, o sistema embebido demorará 5 segundos a perceber que o veículo foi desligado (na secção testes e resultados será referido o erro da distância ao utilizar a fórmula referida acima).
5. Cálculo do consumo do veículo: nesta fase o cálculo do consumo do veículo terá de ser feito com recurso a diversos sensores do automóvel. A forma mais correta de calcular o consumo é através do parâmetro de injeção de combustível (devolve em litros/hora). Assim:
    a. É enviada uma mensagem OBDII com o código do parâmetro de injeção (ID = 5E). Se existir resposta a esta mensagem, o consumo de combustível é retornado de imediato nas unidades litros por hora.
    b. Senão houver resposta à mensagem referida acima, o sistema embebido envia uma mensagem com ID do sensor MAF, e calcula o consumo de combustível da equação (3) representado por *X* (MatthesRieke, 2016):

$$X = \frac{\frac{MAF}{AFR}}{Density\ of\ fuel} \times 3600 = litros/hora \quad (3)$$

c. Senão houver resposta ao sensor MAF, calcula-se o consumo através do sensor MAP (MatthesRieke, 2016):

$$IMAP = \frac{RPM \times MAP}{IAT} \quad (4)$$

$$MAF = \frac{\frac{IMAP}{120} \times \frac{VE}{100} \times ED \times MM}{R} \quad (5)$$

$$X = \frac{\frac{MAF}{AFR}}{Density\ of\ fuel} \times 3600 = litros/hora \quad (6)$$

6. Sincronização dos dados: nesta fase o sistema embebido deverá transmitir de forma segura e fiável as estatísticas do veículo que foram recolhidas enquanto o veículo esteve em funcionamento, sendo que esta sincronização ocorre sempre quando o veículo é desligado.

O cálculo do consumo é algo complexo e que varia dependendo do combustível do automóvel, sendo que será abordado na secção "3.7. Cálculo do consumo de combustível". As equações (3), (4), (5) e (6) serão também detalhadas na mesma secção.

Os parâmetros combustível, cilindrada e quilometragem, em conjunto com o ID do veículo, serão enviados para o sistema embebido para que este se autoconfigure, otimize e adeque o seu funcionamento. Esta autoconfiguração define em que veículo o sistema embebido está ligado e também como a recolha dos diversos dados será feita. Ou seja, como o cálculo do consumo de combustível pode ser realizado de diversas maneiras, o sistema embebido deverá testar qual destas maneiras é possível calculá-lo, devendo

seguir o algoritmo definido no componente "5. Cálculo do consumo do veículo". No sistema estarão guardadas estas configurações num ficheiro que permitem ao mesmo iniciar mais rapidamente, evitando que seja feito as operações no componente "2. Módulo de deteção dos sensores".

3.6.1. Arquitetura do sistema embebido

Cada fase referida anteriormente originou as suas próprias funções, sendo que o diagrama de arquitetura representado pela figura 3.21 pretende demonstrar o fluxo de funcionamento do sistema embebido. Notar que os principais módulos estão diferenciados, e salientar também que um dos módulos representa a obtenção da data e hora atual no sistema embebido (Thread RTC):

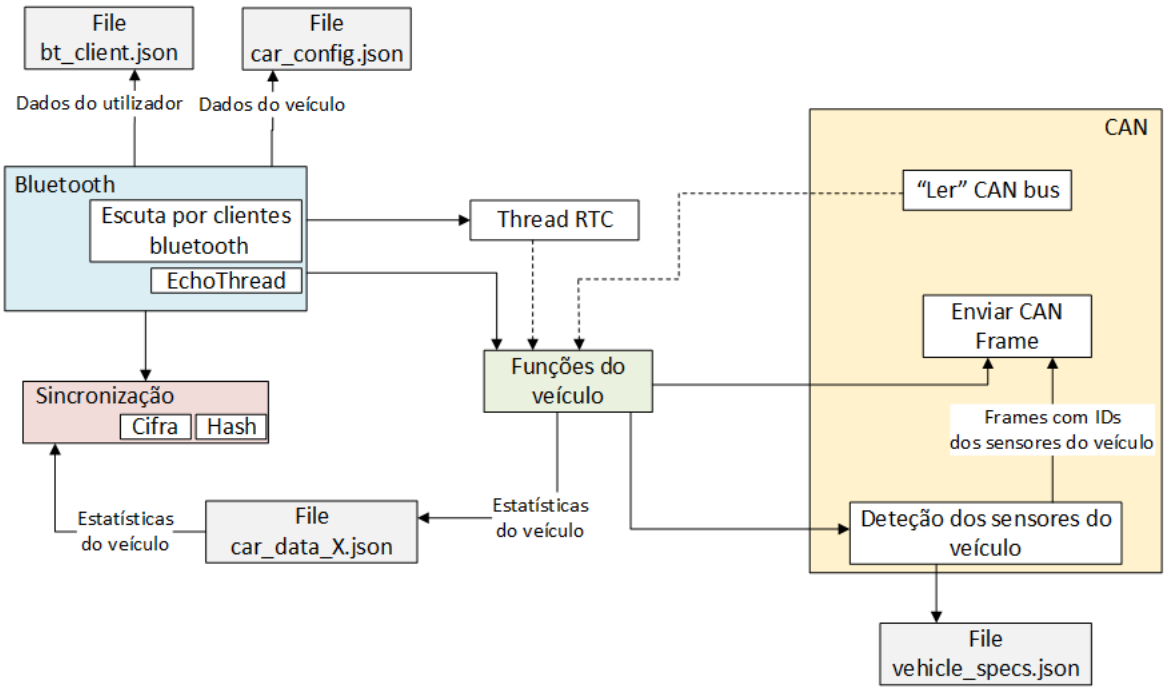


Figura 3.21 – Diagrama de funcionamento do sistema embebido.

3.6.2. Módulo Bluetooth

Este módulo representado por "Escuta por clientes bluetooth" e "EchoThread" tem como objetivo gerir a ligação Bluetooth iniciada pelo dispositivo móvel, e também enviar os dados para o dispositivo móvel, sendo *aware* do estado da

ligação Bluetooth entre o sistema embebido e dispositivo móvel. Este módulo é o primeiro a ser iniciado, pois o cliente deverá iniciar uma ligação Bluetooth ao sistema embebido, de forma que seja guardado num ficheiro ("bt_client.json"): o endereço MAC do cliente, o seu ID de utilizador e a chave simétrica de cifra. Após esta ligação ser estabelecida, o dispositivo móvel deverá enviar para o sistema embebido a configuração do veículo registado (guardada também no ficheiro "car_config.json"), sendo que só após estas duas configurações iniciais é que o sistema embebido inicia a recolha dos dados de veículo onde está ligado (Dostál, 2010).

É este módulo que controla os comandos que são enviados pelo utilizador, sendo eles os seguintes: configuração do veículo – que contém o ID do veículo, ID do utilizador, tipo de combustível, cilindrada, quilometragem; e RESET que apaga qualquer configuração referente ao utilizador e veículo onde o sistema embebido está ligado.

Para garantir a segurança no acesso ao sistema embebido, apenas o utilizador que configurou primeiramente este, é que consegue efetuar o RESET e também alterar as configurações do veículo. Para o acesso ao sistema embebido o utilizador tem sempre de efetuar o login na aplicação móvel, sendo que quando se iniciar a ligação a este, são enviados o ID de utilizador e uma chave de cifra. Certamente que como qualquer utilizador que tenha a aplicação móvel pode tentar ligar-se a qualquer sistema embebido através do Bluetooth, apenas o utilizador autorizado é que irá receber os dados e poder realizar operações sobre este, sendo que as restantes ligações não autorizadas serão terminadas. Para garantir também a privacidade da transmissão dos dados estatísticos do veículo entre o sistema embebido e aplicação móvel, a mesma será cifrada utilizando uma chave simétrica de 32 bytes (referida anteriormente como "chave de cifra") gerada automaticamente aquando do registo do utilizador.

3.6.3. Módulo Funções do veículo

É neste que após o processamento do tráfego OBDII são recolhidos os dados úteis que servem para o cálculo da quilometragem do veículo e consumo de combustível. Aqui são aplicados os algoritmos definidos tanto para os veículos a gasolina como diesel. Este módulo irá detetar também certos comportamentos do próprio veículo (como exemplo paragem do mesmo, ligar e desligar o motor), e gerar os ficheiros relativos às estatísticas do veículo, controlando também quando deverá ser feita a sincronização entre

o sistema embebido e a aplicação móvel. Ou seja, o módulo "Bluetooth" é que é responsável pelo envio dos dados entre a aplicação móvel e o sistema embebido, mas é este módulo ("Funções do veículo") que define quando é que estes dados estão prontos para serem enviados, invocando posteriormente funções do módulo "Sincronização". Este módulo inicia-se depois do módulo "Bluetooth".

Como todas as configurações são guardadas em ficheiros, o sistema embebido sempre que iniciar, pode já começar a recolha de dados do veículo, mesmo que o cliente não esteja ligado a este através de Bluetooth. Na falta destes dois ficheiros, ou apenas de um deles, o módulo "Funções do veículo" não é iniciado e, por conseguinte, o sistema embebido aguarda por configurações provenientes da aplicação móvel.

3.6.4. Módulo CAN

Este módulo representado por "Deteção dos sensores do veículo", "Ler CAN bus" e "Enviar CAN Frame" contém funções cujo objetivo é controlar todo o tráfego que é gerado no barramento de veículo, ao processar apenas o tráfego de interesse, sendo apenas iniciado assim que o módulo "Funções do veículo" o invocar. É numa função deste módulo que é feita a verificação de que sensores o veículo possui, para que depois estes possam ser utilizados mais tarde no cálculo, tanto da distância como do consumo do veículo. Os valores dos sensores são escritos posteriormente num ficheiro ("vehicle_specs.json"), para que no arranque do sistema, o sistema embebido possa já carregar os valores referentes aos sensores do veículo sem que seja necessário tornar a questionar o veículo de quais sensores este possui. Também já foi referido o facto de que não serão geradas novas *frames* OBDII enquanto não forem recebidas as respetivas respostas, independente do tempo de demora destas respostas. Com esta forma protege-se o veículo, pois minimiza-se a quantidade de tráfego gerado no barramento CAN através da interface de diagnóstico.

3.6.5. Módulo Sincronização

Este módulo tem como objetivo enviar os dados para a aplicação móvel, devendo conhecer todos os ficheiros referentes às estatísticas do veículo que já foram enviados e os que ainda não foram enviados, garantir a integridade destes dados aquando da sua receção pela aplicação móvel, e reenviá-los caso seja necessário. O nome dos

ficheiros gerados pelo módulo "Funções do veículo" seguem uma regra, sendo sempre "car_data_X.json", e cujo "X" é valor da data e hora no momento, fornecida pela "Thread RTC". A cada dois minutos de viagem é gerado um novo ficheiro, e com esta nomenclatura evitam-se nomes de ficheiros duplicados. Este módulo possui funções que permitem que os dados transmitidos sejam rececionados de forma íntegra (*hashing* – SHA256) (Dwernychuk, 2017) e de forma segura (através da cifra AES) (Github, 2017).

A sincronização é feita de forma automática em que o sistema embebido envia os dados quando o veículo é desligado, ou seja, quando as rotações por minuto têm o valor zero. Contudo, para não sobrecarregar o barramento CAN, a *frame* gerada para validar as rotações por minuto do motor, é enviada apenas de cinco em cinco segundos, e por isso, o sistema embebido no pior cenário poderá demorar estes cinco segundos a perceber que o veículo foi desligado.

3.7. Cálculo do consumo de combustível

O cálculo de combustível do veículo (MatthesRieke, 2016) foi uma das fases no desenvolvimento do software para sistema embebido, porque apesar de que através do standard OBD saberem-se inúmeros parâmetros relativos ao automóvel, no que que diz respeito aos consumos de combustível instantâneos não é possível por vezes obter este valor diretamente. Por isso, terá de ser calculado de uma das seguintes formas:

- Através do parâmetro *Engine Fuel Rate*
- Através dos sensores:
    - MAF (*mass air flow*)
    - MAP (*manifold absolute pressure*)

O parâmetro "*Engine Fuel Rate"*, mede o combustível que é injetado nos cilindros de automóvel num determinado momento. A unidade do automóvel calcula este valor através do débito da bomba de injeção e do tempo de abertura dos injetores, aquando da injeção de combustível. O PID OBDII que retorna valor deste sensor é o 5E, e as unidades são l/h. Com o somatório das leituras deste parâmetro e da velocidade (que servirá para calcular a distância percorrida), consegue-se obter uma média de consumo de litros de combustível aos 100 quilómetros.

A equação (7) demonstra este cálculo:

$$\frac{Somatório(EFR) \times 100}{Somatório(distância\ perccorida)} = litros/100Km \quad (7)$$

A equação (7) acima aplica-se a todos os automóveis, sejam a gasolina ou diesel. Contudo quando este parâmetro não existe é necessário recorrer aos sensores MAF ou MAP.

Tanto o sensor MAF como o MAP têm a mesma função, que é de medir a quantidade de ar que entra na admissão do veículo. Abaixo segue-se uma breve explicação de como cada um destes sensores funciona:

- MAF – *mass air flow*, este sensor utiliza um fio aquecido para calcular a quantidade de ar que entra no motor. Nos veículos a gasolina, a razão entre ar e combustível (AFR – *air to fuel ratio*) tende a ser em média próxima do valor ideal, mas nos veículos a diesel o mesmo já não acontece. Este sensor envia um sinal digital para a ECU (*Engine Control Unit*) que traduz a quantidade de ar que está a passar por ele. A ECU utiliza este sinal para calcular a quantidade de combustível a injetar nos cilindros de modo a manter o AFR no nível adequado e manter o nível de emissões de gases poluentes baixo, sem sacrificar a performance e longevidade do motor (Samarins, 2018).
- MAP – *manifold absolute pressure*, sensor este que também tem a função de medir a quantidade de ar que entra na admissão do motor, mas que adicionalmente é necessário saber a cilindrada do motor e o máximo de RPMs (rotações por minuto). Este sensor gera um sinal que é proporcional ao vácuo no coletor de admissão. A ECU ajusta o tempo de ignição e a quantidade de combustível a injetar. Quando o motor não está a trabalhar, a pressão no coletor de admissão é a mesma que a pressão no exterior. Quando o motor arranca, um vácuo é criado pelo próprio movimento dos pistons. Quando acelerado ao máximo, o vácuo "cai" para próximo de 0, ficando a pressão quase igual à pressão no exterior. A ECU com o valor do sensor MAP, rotação do motor, posição do acelerador, temperatura do motor e ambiente, estima a quantidade de ar que entra no motor. Além disto, a ECU poderá ter em conta o valor do sensor de oxigénio (sonda lambda) presente

na linha de escape, antes de aplicar as correções na relação entre ar e combustível de modo a manter todo o sistema balanceado (AA1Car, s.d.).

3.7.1. Veículos a gasolina

Um veículo a gasolina, tenta manter as suas reações de combustão próximas duma combustão ideal, sendo que o facto de existir uma válvula, que ao abrir e fechar consoante o acelerador altera o valor de admissão de ar e por conseguinte a unidade de controlo do motor altera a injeção de combustível. Ou seja, os valores de combustível e ar são variáveis.

A equação (8) representa o cálculo do consumo de combustível utilizando o sensor MAF (MatthesRieke, 2016):

$$F = \frac{MAF}{AFR \times Densidade\ do\ combstível} \times 3600 = litros/hora \qquad (8)$$

Em que MAF é o valor do sensor de massa de ar (medida em gramas por segundo), e o AFR corresponde ao *air to fuel ratio*, que é a razão entre a quantidade de ar e quantidade de combustível, de forma que esta reação de combustão seja estequiométrica. Uma reação estequiométrica é uma reação onde a quantidade dos elementos reagentes é igual à quantidade desses mesmos elementos no produto da combustão, e onde o valor das emissões de poluentes é menor (exceto os valores de *NOx*, em que combustões perto de AFR ideal, produzem temperaturas mais elevadas aumentando a formação destes gases). Para o caso de motores a gasolina, o AFR ideal é de 14.7:1, ou seja, 14.7 quilogramas de ar por 1 quilograma de combustível. Já para motores a diesel, o AFR é variável, mas o valor ideal é 14.5. Assim em veículos a gasolina, para que a combustão seja ideal, são necessários 14,7Kg de ar para se consumir 1Kg de gasolina.

Na equação (8), a utilização de uma AFR constante deve-se ao facto de num veículo a gasolina a média do valor de AFR de todos os ciclos de combustão ser próxima do valor do AFR ideal. Um veículo a gasolina, como controla a admissão de ar e injeção de combustível pode conseguir manter as combustões num AFR próximo do ideal. Contudo na prática estes motores não devem trabalhar continuamente num AFR ideal

pois é também nestas condições que a temperatura pode aumentar ao ponto de danificar o motor, tendo a própria gasolina a função de arrefecer a câmara de combustão. Embora a formação de determinados poluentes seja maior numa AFR perto do ideal, a combustão com valores de AFR altos (misturas pobres), para motores a gasolina reduz a potência gerada e aumenta a probabilidade *engine knocking* (detonações prematuras) (Amir Khajepour, 2014).

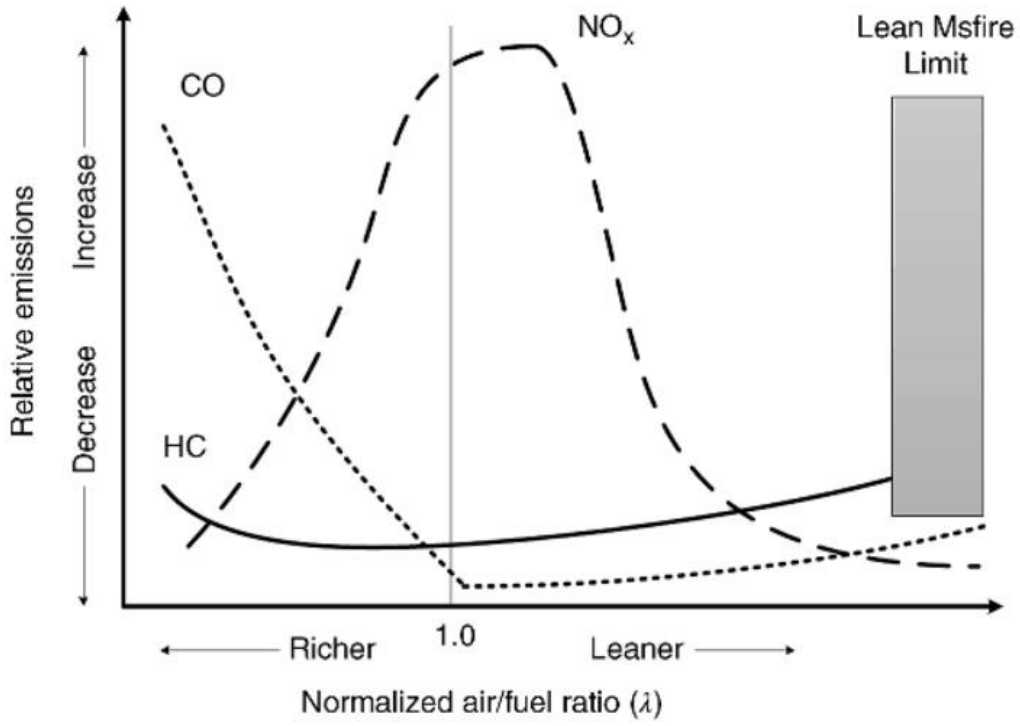


Figura 3.22 – Efeito do AFR no valor de emissões de poluentes para um motor a gasolina, Adaptado (Amir Khajepour, 2014).

Como é visível na figura 3.22, o valor de emissões dos gases NOx é maior em combustões perto do AFR ideal ($\boldsymbol{\lambda} = 1$), e em misturas pobres (*leaner*) a emissão de qualquer poluente é menor. Como já foi referido anteriormente, misturas pobres potenciam a probabilidade de *engine knocking* e geram menos potência.

A figura 3.23 traduz o valor de AFR para cada regime do motor, em função do binário gerado e rotação por minuto do motor, num veículo a gasolina, sendo que cada modelo e versão de veículo tem a sua própria tabela de AFR:

Engine speed [rpm]

Engine torque [Nm]	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500
10	14	14.7	16.4	17.5	19.8	19.8	18.8	18.1	18.1	18.1	18.1	18.1	18.1
20	14	14.7	14.7	16.4	16.4	16.4	16.5	16.8	16.8	16.8	16.8	16.8	16.8
30	14	14.7	14.7	14.7	14.7	14.7	14.7	15.7	15.7	15.3	14.9	14.9	14.9
40	14.2	14.7	14.7	14.7	14.7	14.7	14.7	14.7	14.7	13.9	13.3	13.3	13.3
50	14.7	14.7	14.7	14.7	14.7	14.7	14.7	14.7	14.7	14.5	12.9	12.9	12.9
60	14.7	14.7	14.7	14.7	14.7	14.7	14.7	14.7	14.3	13.3	12.6	12.1	11.8
70	14.7	14.7	14.7	14.7	14.7	14.7	14.7	14.7	13.6	12.9	12.2	11.8	11.3
80	14.1	14.2	14.7	14.7	14.7	14.7	14.7	14.7	13.3	12.5	11.9	11.4	10.9
90	13.4	13.4	13.8	14.3	14.3	14.7	14.7	13.6	13.1	12.2	11.5	11.1	10.7
100	13.4	13.4	13.4	13.4	13.4	13.6	13.6	12.1	12.1	11.6	11.2	10.8	10.5
110	13.4	13.4	13.4	13.4	13.1	13.1	13.1	11.8	11.8	11.2	10.7	10.5	10.3
120	13.4	13.4	13.4	13.4	12.9	12.9	12.5	11.6	11.3	10.5	10.4	10.3	10.2
130	13.4	13.4	13.4	13.4	12.9	12.9	12.5	11.6	11.3	10.5	10.4	10.3	10.2
140	13.4	13.4	13.4	13.4	12.9	12.9	12.5	11.6	11.3	10.5	10.4	10.3	10.2

Figura 3.23 – Mapa AFR veículo a gasolina, Adaptado (Stark, s.d.).

Tal como o *Engine Fuel Rate*, nem todos os automóveis possuem um sensor MAF. Assim é necessário utilizar o sensor MAP (*manifold absolute pressure*), sendo o seu cálculo representado na equação (9) (MatthesRieke, 2016):

$$F = \frac{MAF}{AFR \times Densidade\ do\ combstível} \times 3600 = litros/hora \qquad (9)$$

Mas como o sensor MAF não está presente, a variável MAF é calculada através do sensor MAP (MatthesRieke, 2016):

$$MAF = \frac{\frac{IMAP}{120} \times \frac{VE}{100} \times ED \times MM}{R} \qquad (10)$$

$$IMAP = \frac{RPM \times MAP}{IAT} \qquad (11)$$

Em que RPM é o valor da rotação por minuto do automóvel, MAP o valor do sensor MAP e IAT a temperatura em Kelvin do ar de admissão (0F é o parâmetro em graus celsius da temperatura do ar de admissão fornecido pelo standard OBD2). VE corresponde à eficiência volumétrica, ou seja, eficiência em conseguir colocar mais ar nos cilindros de combustão, sendo que nos automóveis situa-se entre os 80%-85%. A variável ED corresponde à cilindrada do motor, MM à massa molecular do ar e R à constante universal dos gases ideais.

A densidade do combustível possui os valores de:

- 745 gramas/litro para a gasolina
- 832 gramas/litro para o diesel

3.7.2. Veículos a diesel

Para veículos a diesel o cálculo do consumo de combustível é mais complexo visto que a razão entre ar e combustível (AFR) é variável, e que as misturas num veículo a gasóleo são "pobres" (ou seja, muito pouco combustível para o mesmo ar e AFR mais alto). Num veículo a diesel, a quantidade de ar é a mesma, apenas a injeção de combustível é que varia (Physics Forums, 2017).

Mas, tal como os veículos a gasolina, os veículos a diesel podem também possuir os sensores EFR, MAF ou MAP. Assim, as fórmulas para o cálculo de combustível utilizadas serão as mesmas que as utilizadas para os veículos a gasolina, mas para o caso do cálculo através dos sensores MAF ou MAP é necessário definir a correta densidade do combustível (gasóleo neste caso) e o valor de AFR (que é variável). Na secção "Trabalho futuro" será discutida uma abordagem que explicará o desenho duma tabela de AFR genérica para os motores a diesel.

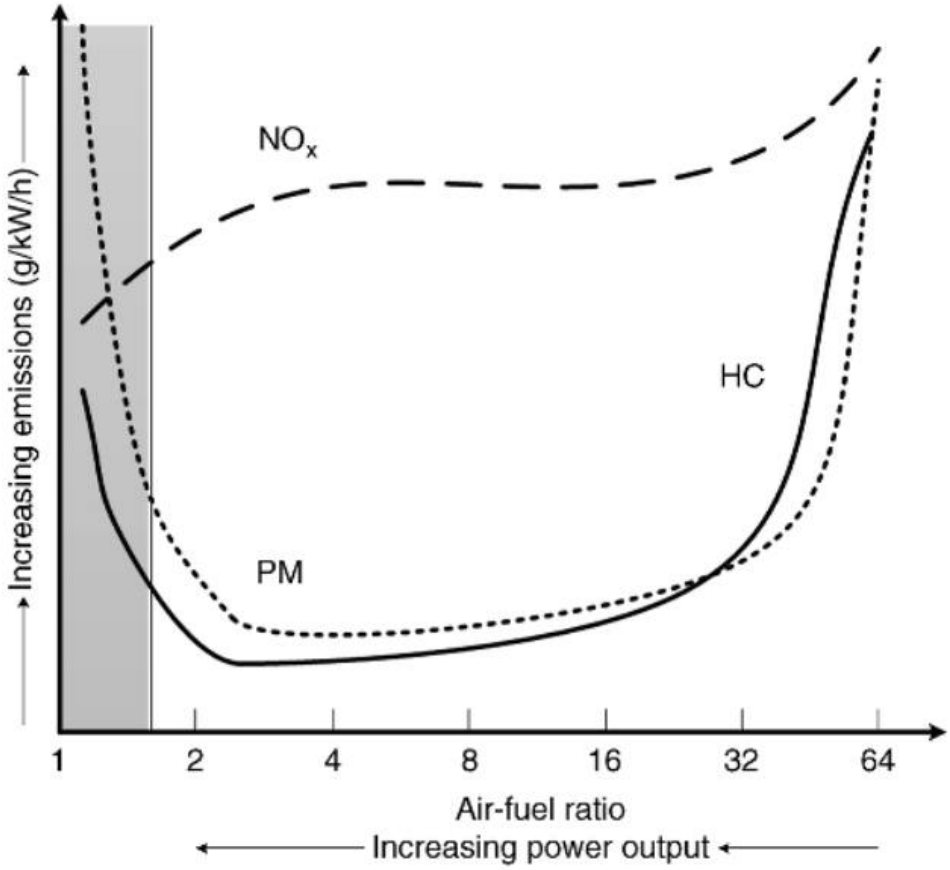


Figura 3.24 – Efeito do AFR no valor de emissões de poluentes para um motor a diesel, Adaptado (Amir Khajepour, 2014).

Os veículos a diesel emitem principalmente partículas (PM), óxidos nitrosos (*NOx*) e hidrocarbonetos (HC). Como a figura 3.24 ilustra, quando o AFR tem um valor elevado (mistura pobre) ou muito baixo (mistura rica), a emissão destes poluentes é maior. Como se visualiza também, a formação de NOx é sempre elevada para os vários valores de AFR (ou seja, para os vários regimes do motor a diesel) (Amir Khajepour, 2014).

3.8. Protótipo do sistema embebido

A figura 3.25 mostra os diferentes componentes de hardware do sistema embebido do OwnGarage: RTC, Raspberry Pi e Controlador CAN. É possível também visualizar o cabo necessário para ligar o sistema embebido ao automóvel através da interface de diagnóstico.

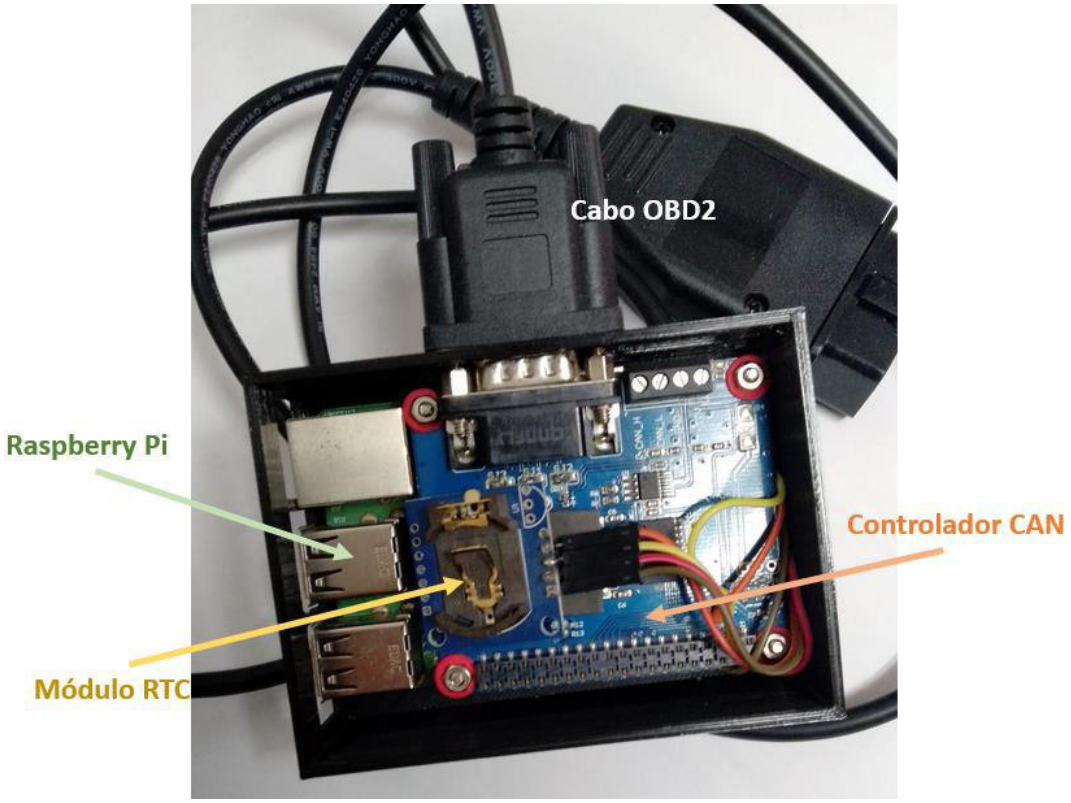


Figura 3.25 – Protótipo sistema embebido do OwnGarage.

Foi utilizada uma impressora 3D para construir a caixa que engloba todo o hardware do sistema embebido. A caixa tem as dimensões de 9.2 x 6.8 x 2.4 (unidades em centímetros), terá um cabo OBDII com conversão para RS232 para comunicação com a interface de diagnóstico do automóvel, e será alimentado através da tomada interna de 12 Volts do automóvel (com um conversor de 12 Volts para 5 Volts).

4. TESTES E RESULTADOS

Este capítulo contém os resultados obtidos aquando do teste dos vários protótipos finais de cada subsistema do OwnGarage.

4.1. Testes com o sistema embebido e automóvel

O teste efetuado entre o sistema embebido e o automóvel foi o cálculo da quilometragem do mesmo. Este cálculo é necessário pois o OwnGarage necessita do valor da quilometragem do automóvel para poder notificar os utilizadores da proximidade das tarefas de manutenção. Este cálculo foi realizado com recurso ao valor sensor de velocidade do automóvel, valor este que é obtido através do PID 0D definido no standard OBD.

O veículo utilizado nos testes foi um Opel Astra H do ano 2005, cuja motorização é 1.7 CDTI, combustível gasóleo.

Durante este teste, em 26 quilómetros percorridos o erro foi de 16 metros. O que significa que o veículo mostrava no painel de instrumentos 26 quilómetros percorridos, e o sistema embebido calculou 25.984 Km. Durante esta fase de testes, foram feitas várias paragens, e detetou-se que durante esta viagem, o erro era sempre aproximadamente o mesmo, ou seja, é um erro aproximadamente constante e não incremental. Existe este erro porque o cálculo da quilometragem do automóvel é feito através duma média em função das amostras de velocidade recolhidas e o intervalo tempo que foram recolhidas.

Em relação ao sistema embebido e o veículo, não foram feitos mais testes porque num destes testes, aquando da imobilização do veículo, detetou-se uma luz no painel de instrumentos indicando avaria no sistema de controlo de estabilidade. Concluímos que este veículo não possui qualquer sistema de segurança no controlo de acesso ao barramento CAN por OBDII, e por uma questão de segurança, foi decidido que não seriam feitos mais testes no veículo com o sistema embebido.

4.2. Testes com sistema embebido e aplicação móvel

Para os testes entre o sistema embebido e aplicação móvel, foi decidido efetuar dois testes:

- Teste de velocidade de transmissão dos dados recolhidos pelo sistema embebido.
- Teste de autonomia de bateria do dispositivo móvel.

4.2.1. Teste de velocidade de transmissão

Foi realizado um teste de velocidade de transmissão dos dados entre o sistema embebido e a aplicação móvel Android. Como referido anteriormente, a solução passava por gerar um ficheiro cada vez que o veículo era ligado e desligado, ficheiro esse que continha as estatísticas do veículo no intervalo de tempo que este esteve em funcionamento.

Durante o teste verificou-se que não era possível enviar ficheiros dum tamanho elevado, pois a própria biblioteca em Python, correspondente à comunicação Bluetooth, segmentava este pacote a ser transmitido, gerando depois erro na aplicação móvel. Por isso poderá abordar-se das duas seguintes formas:

- Segmentar o ficheiro gerado por cada viagem do veículo.
- Gerar múltiplos ficheiros para a mesma viagem.

Para este teste, foi utilizada a abordagem definida no segundo ponto, sendo gerado um ficheiro a cada 2 minutos da viagem. Também, como não é possível efetuar estes testes no veículo, foi gerado um ficheiro com dados aleatórios das estatísticas do veículo, referente a sessenta minutos de viagem. Na secção "Trabalho futuro" será explicado o funcionamento de cada uma destas abordagens.

Assim, para 1 hora de viagem são gerados trinta ficheiros diferentes e existe sempre a confirmação da receção correta para cada ficheiro. Ou seja, sempre que é enviado um ficheiro para a aplicação móvel, esta deverá validar se os dados foram recebidos corretamente e enviar esta mesma resposta ao sistema embebido, para que este envie o próximo ficheiro. Cada ficheiro recebido corretamente é guardado na memória

interna do dispositivo móvel. A figura 4.1 representa um gráfico do tempo despendido pelo dispositivo móvel no processamento de cada pacote que recebe do sistema embebido.

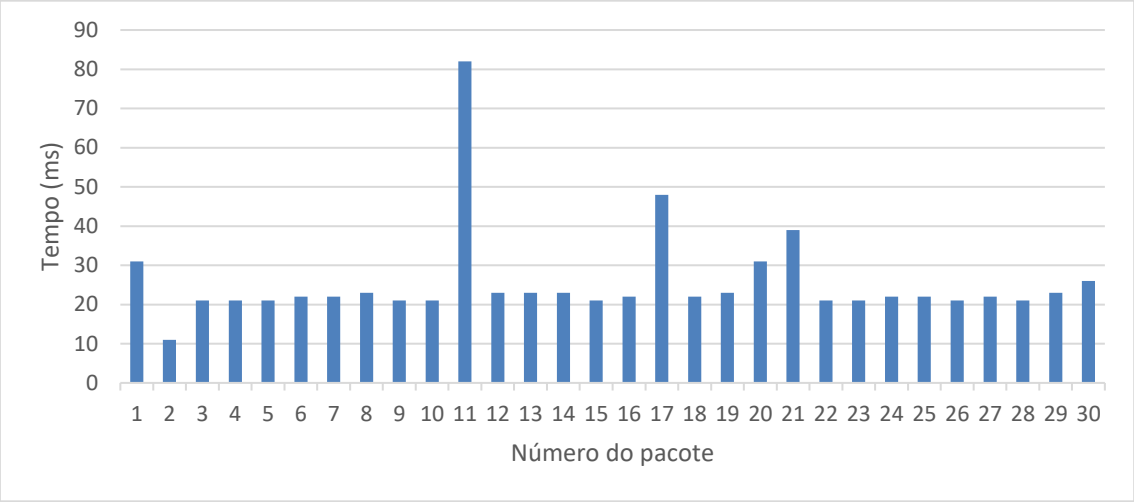
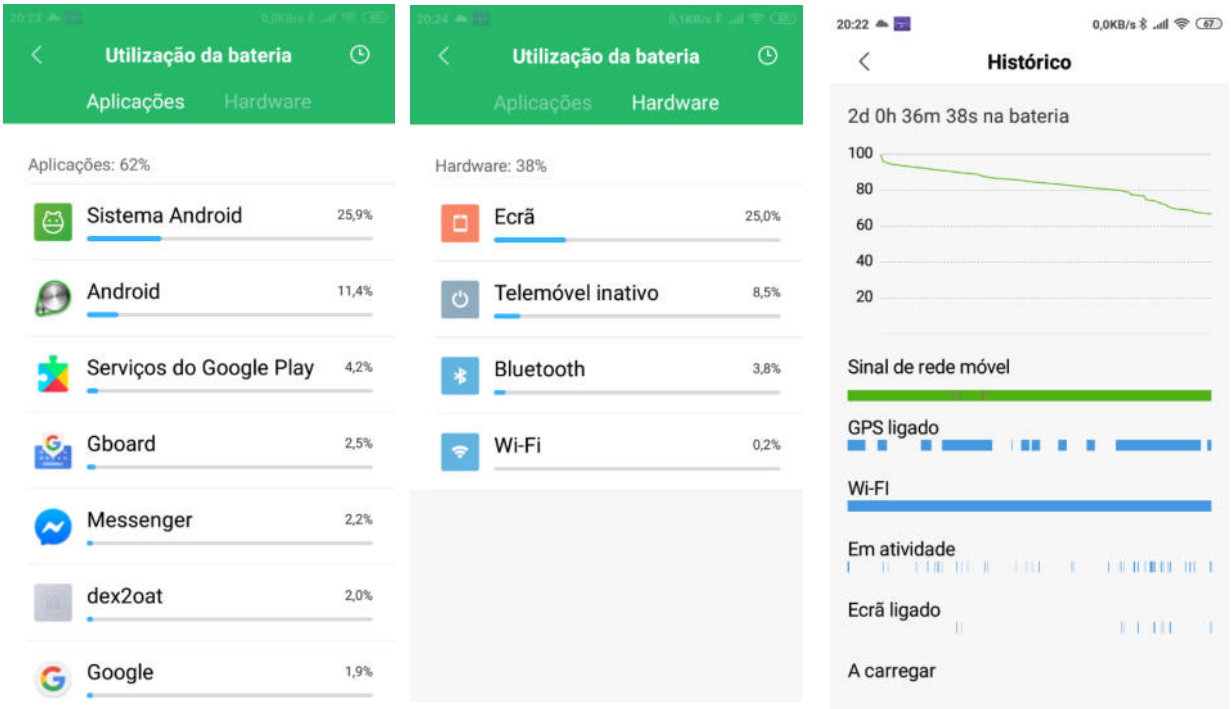


Figura 4.1 – Gráfico de tempo de processamento dos pacotes entre o dispositivo móvel e o sistema embebido.

Assim, para uma viagem de uma hora, o tempo gasto para o envio dos ficheiros referentes a esta viagem foi de aproximadamente um 1 segundo, sendo que o tempo médio de processamento de cada ficheiro (entenda-se este processamento desde o envio do ficheiro, decifra dos dados, cálculo de *checksum* para validação da integridade, e receção do resultado da validação anterior) situa-se entre os 11 milissegundos e os 82 milissegundos. Pelos valores do gráfico da figura 4.1, a média de processamento de cada pacote é de 22 milissegundos. Mas, se contarmos com os 5 segundos que o sistema embebido pode demorar a perceber que o veículo foi desligado, temos que o tempo de transmissão poderá ser no máximo 6 segundos. Mesmo com este valor concluiu-se que para 1 hora de viagem, o tempo é aceitável para a sincronização destes dados.

4.2.2. Teste de autonomia de bateria

O segundo teste efetuado entre a aplicação móvel e o sistema embebido foi o teste sobre o impacto da aplicação móvel na autonomia de bateria do dispositivo móvel. A figura 4.2 mostra diferentes imagens relativas a estatísticas do consumo de bateria do dispositivo móvel quando este tem a aplicação OwnGarage instalada, já a figura 4.3 mostra o consumo de bateria do dispositivo móvel sem a aplicação do OwnGarage instalada. As estatísticas foram recolhidas através do sistema operativo móvel Android, em que o teste teve a duração de aproximadamente 2 dias e o telemóvel utilizado foi um Xiaomi Redmi 3S com versão de Bluetooth 4.1. Para simular uma utilização normal por parte dum utilizador final, neste teste foram realizadas 20 sincronizações entre o dispositivo móvel e o sistema embebido. Em cada uma destas sincronizações são transmitidos os dados referentes a uma hora de viagem.



a) b) c)

Figura 4.2 – a) Separador "Aplicações" – com aplicação instalada
b) Separador "*Hardware*" – com aplicação instalada
c) Separador "Histórico" – com aplicação instalada.

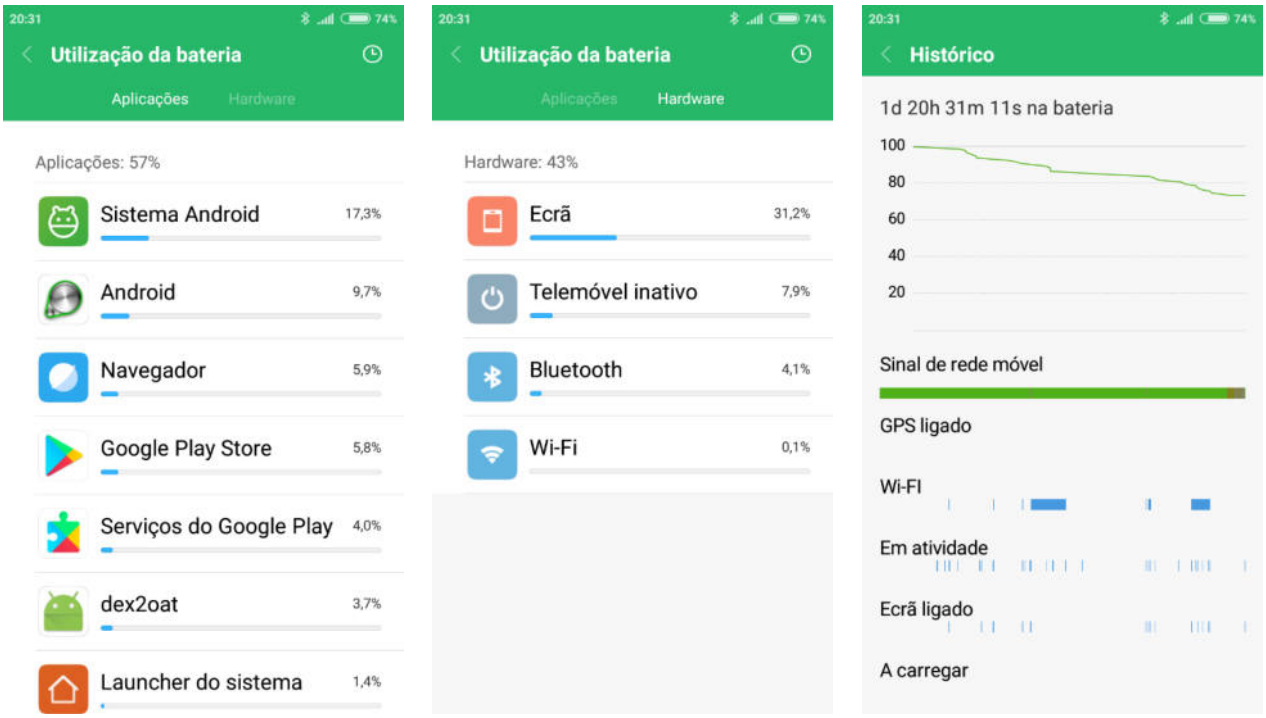


Figura 4.3 – a) Separador "Aplicações" – sem aplicação instalada
b) Separador "*Hardware*" – sem aplicação instalada
c) Separador "Histórico" – sem aplicação instalada.

Tanto na figura 4.2 e 4.3 a alínea a) representa as aplicações que mais consomem bateria ordenadas por ordem crescente de consumo; a alínea b) o hardware responsável por consumir mais bateria, ordenado também por ordem crescente de consumo; e a alínea c) o histórico do consumo de bateria desde que a mesma foi carregada até à data que se finalizou o teste.

Se compararmos a alínea a) da figura 4.2 e figura 4.3 visualizamos que aplicação OwnGarage tem um impacto mínimo no consumo de bateria, pois nem sequer a mesmo é visível nesta figura.

Se compararmos a alínea b) da figura 4.2 e 4.3 também visualizamos que o consumo de bateria por parte do hardware Bluetooth não é notório, muito devido ao facto da versão do mesmo ser 4.1 e também de serem necessários apenas 1 segundo para sincronizar os dados de uma hora de viagem.

Por fim, ao analisarmos a alínea c) de cada uma das figuras, visualiza-se que consumo de bateria não é excessivo, sendo que para o mesmo tempo de teste, a percentagem de bateria restante é de:

- 74% quando o dispositivo móvel não tem aplicação OwnGarage instalada
- 67% quando o mesmo tem a aplicação OwnGarage instalada.

Perante estes valores de consumo de bateria, concluímos assim que a aplicação móvel OwnGarage, num uso diário do dispositivo móvel, não põe em causa a autonomia da bateria do mesmo.

5. CONCLUSÕES

Com a tecnologia nos dias de hoje os dispositivos IoT são de uma importância vital e que facilmente permitem aos utilizadores desenvolver produtos capazes de interligar vários sistemas diferentes entre si, de forma simples e de alto nível. Verifica-se que a indústria automóvel está também a evoluir no sentido de acompanhar esta evolução tecnológica, apresentando aos utilizadores componentes modernos cuja contribuição vai desde o conforto dos passageiros até à gestão do motor e segurança do veículo (seja ativa ou passiva). Alguns dos fabricantes já disponibilizam aos seus clientes um serviço em que uma aplicação móvel pode dar informações pertinentes sobre o estado do veículo do utilizador, estando isto disponível para automóveis recentes.

O projeto OwnGarage visa fornecer uma experiência semelhante a estas aplicações, mas que ao mesmo tempo consiga abranger veículos mais antigos, auxiliando os utilizadores na gestão das tarefas de manutenção dos seus automóveis, e para as oficinas a gestão de tempo das suas reparações. O sistema é constituído por vários módulos, cada um capaz de fornecer funcionalidades específicas aos utilizadores, e que em conjunto contribuem para esta maior facilidade na gestão das manutenções dos automóveis.

Ao ser implementado desta forma conseguiu-se ter um sistema com resultados positivos, embora o sistema embebido tenha apresentado vários problemas na fase de testes com o automóvel. Ou seja, estes resultados mostraram que a utilização do sistema embebido não é segura quando este está ligado através da interface de diagnóstico ao veículo, porque o veículo utilizado para este efeito, não possui quaisquer sistemas de segurança no acesso ao barramento CAN através desta interface. Mas no geral, este sistema consegue expor de forma fácil e simples as suas funcionalidades, sendo um dos pontos positivos, quando comparado com outros projetos similares, o facto de não ser necessário uma conexão sempre ativa entre o sistema embebido e o dispositivo móvel, visto que este primeiro recolhe a informação no modo "*offline*" (entenda-se este modo como "*disconnected*" do dispositivo móvel), e sincroniza estes dados recolhidos assim que um dispositivo móvel válido se ligar a este. Como este sistema utiliza a tecnologia Bluetooth, é fácil para um utilizador não intencionado, tentar ganhar acesso ao sistema

embebido que está ligado ao automóvel, e por isso o OwnGarage além de ser capaz da cifra dos dados transmitidos entre este módulo e o dispositivo móvel, dispõe também de funções que permitem o controlo de acessos ao sistema embebido.

O projeto não possui funcionalidades que sejam capazes do cálculo do consumo de combustível, ficando esta implementação para trabalho futuro. O porquê de não terem sido implementadas, deveu-se ao facto de os testes no veículo terem originado problemas relativos à segurança do automóvel. É de salientar que apesar destes constrangimentos, com o teste de envio dos dados referentes às viagens entre o sistema embebido e o dispositivo móvel, atingiram-se tempos de transmissão bastantes favoráveis para a utilização deste sistema no quotidiano (o teste é apenas referente a 1 hora de viagem).

Por fim conclui-se que este sistema possui bastantes pontos fortes que fazem dele uma boa solução para as tarefas de manutenção dos veículos e para a gestão das reparações nas oficinas, interligando oficinas e centralizando todas as reparações e estatísticas pertencentes a cada veículo registado.

5.1. Trabalho Futuro

Pretende-se no futuro que todo este sistema possua um registo detalhado de tudo o que os utilizadores realizam sobre o mesmo. Isto é feito com recurso a uma tabela nova na base de dados, contendo todos os registos de operações que sejam efetuadas sobre o sistema, sendo que, como foi explicado anteriormente, as aplicações web e móvel têm de obrigatoriamente aceder a *webservices* na Internet, que por sua vez têm associadas funções presentes na base de dados (de forma a evitar o não repúdio – o quê, quem e quando).

No futuro espera-se que cada fabricante de automóveis possa expor uma API que os utilizadores possam utilizar de forma segura (poderá haver o pagamento de *royalties* para usufruir destes serviços) para a recolha de dados estatísticos (e até mesmo de diagnóstico) dos seus veículos. Como já foi referido e concluído na fase de testes, o método utilizado para conseguir recolher métricas do veículo do utilizador não é o mais seguro, devendo-se principalmente porque o sistema embebido transmite dados num

barramento destinado a diagnóstico automóvel, e barramento esse onde circulam também dados dos sistemas de segurança ativa e passiva.

Pretende-se melhorar a gestão da comunicação entre o sistema embebido e a aplicação móvel, pois embora neste momento o sistema embebido não permita que um utilizador diferente do que está atualmente configurado se ligue ao mesmo, permite que caso o utilizador troque de dispositivo móvel continue a ter acesso a este. A melhoria é relativa ao problema em que caso o utilizador se ligue ao sistema embebido através de mais de um dispositivo móvel, qual o dispositivo móvel é que deverá receber os dados estatísticos do veículo.

Para o cálculo do consumo de combustível dos veículos, em relação à gasolina, apenas não foi implementado este cálculo no sistema embebido, embora a forma como é calculado este valor esteja definida neste relatório. Não foi implementado, devido ao facto dos testes no veículo utilizado terem originado problemas.

Nos veículos a diesel, as fórmulas de cálculo não são tão diretas, e por isso será necessário desenhar uma tabela de AFR personalizada, que será baseada na figura 5.1 que representa o mapa da sonda lambda registada num veículo a diesel. Notar que o valor da sonda lambda depende do AFR ideal e do AFR atual (Stark, s.d.), tal como na equação (12) (Stark, s.d.):

$$\lambda = \frac{\mathrm{AFR}_{\mathrm{actual}}}{\mathrm{AFR}_{\mathrm{ideal}}} \qquad (12)$$

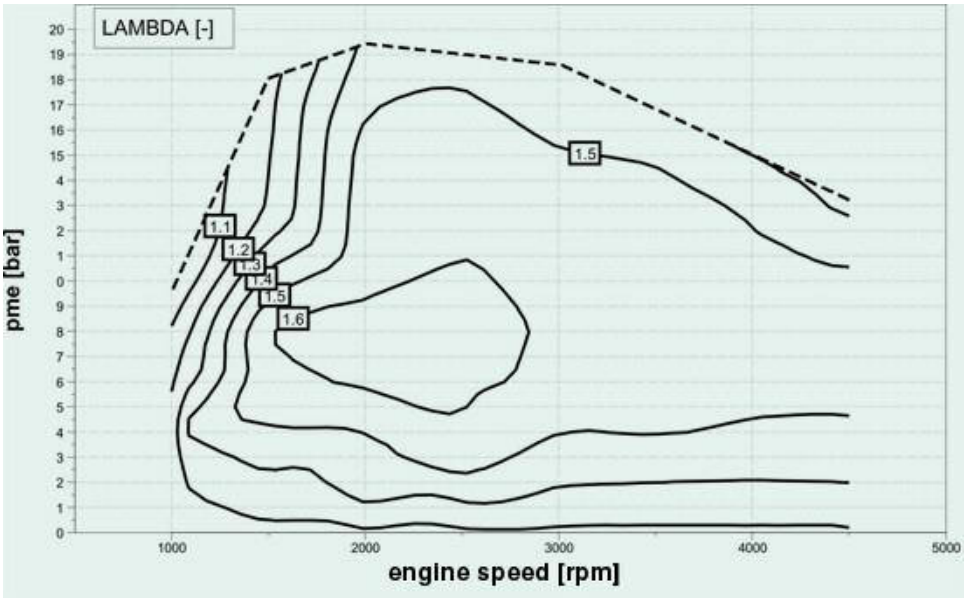


Figura 5.1 – Mapa da sonda lambda motor a Diesel, Adaptado (Stark, s.d.).

A figura 5.1 representa o valor da sonda lambda (ou de oxigénio) para os vários regimes do motor do veículo, em função da rotação do motor e a pressão média efetiva (é um parâmetro utilizado para medir a performance dum motor de combustão interna, podendo relacionar-se com a pressão média no cilindro para um ciclo de combustão completo). A tabela de AFR genérica seria completada com os vários valores de AFR, cujos eixos de XX e YY seriam respetivamente a rotação do motor e a posição, em percentagem, do acelerador do veículo (parâmetro que pode também ser recolhido através de OBDII).

No sistema atual, no sistema embebido não existe um controlo de erros nos ficheiros quando por exemplo há uma quebra de energia neste módulo. Isto pode resultar em que estes ficheiros de estatísticas fiquem corrompidos. Por isso, no futuro seria desenvolvida uma função que corrigisse os ficheiros de forma a que no caso de uma falha de energia não fossem perdidos os dados estatísticos da viagem do utilizador.

Na secção "Testes e resultados" foi referido um problema na transmissão de dados entre o sistema embebido e a Aplicação móvel quando estes pacotes a transmitir tinham um tamanho mais elevado. Assim as duas abordagens para resolver este problema são:

- **Segmentação do pacote**: cada pacote JSON enviado contém os campos: identificador, dados e HASH dos dados, sendo que o campo "dados" (que corresponde às estatísticas do veículo) possui o maior tamanho. Nesta implementação, o campo "dados" seria segmentado, e gerado um ficheiro JSON (na mesma com o identificador e HASH) e cujo campo "dados" iria possuir um segmento do campo "dados" total. No fim do campo HASH, seria incluído um campo denominado de "contador" que serviria para controlo da receção do pacote na totalidade por parte da aplicação móvel. Este contador seria decrementado até chegar a zero, que correspondia há não existência de mais nenhum segmento a ser recebido. Estas regras são similares ao protocolo TCP utilizado atualmente nas redes de computadores.
- **Gerar múltiplos pacotes para a mesma viagem**: a cada dois minutos é gerado um novo ficheiro, que contém as estatísticas referentes a dois minutos da viagem do veículo. Esta forma é mais simples de implementar, destacando apenas a necessidade de ter uma função que faça a gestão dos pacotes existentes e os que já foram enviados e recebidos corretamente pela aplicação móvel. Como é descrito nos resultados, esta solução até pode ser mais viável, pois por extrapolação do teste realizado, se o tempo de transmissão dos ficheiros referentes a oito horas de viagem for de oito segundos, poderá acontecer o caso que o utilizador se ausente do veículo ainda antes de todos os ficheiros serem recebidos, tendo sido apenas enviado parte dos ficheiros, ou seja, parte das estatísticas da referida viagem. Se somarmos o tempo máximo que o sistema embebido pode levar até perceber que o veículo foi desligado (cinco segundos), este tempo total de transmissão duma viagem de 8 horas poderá ser de treze segundos. Assim, mesmo com este intervalo de tempo, gerar múltiplos ficheiros é uma solução mais viável, pois a sincronização dos dados recolhidos poderá ser parcial, ao invés do método de segmentação do pacote que poderá ser nula.

Por fim, a arquitetura do OwnGarage teria de ser alterada segundo a figura 5.2, de forma a criar uma nova funcionalidade que traz comodidade a clientes e oficinas:

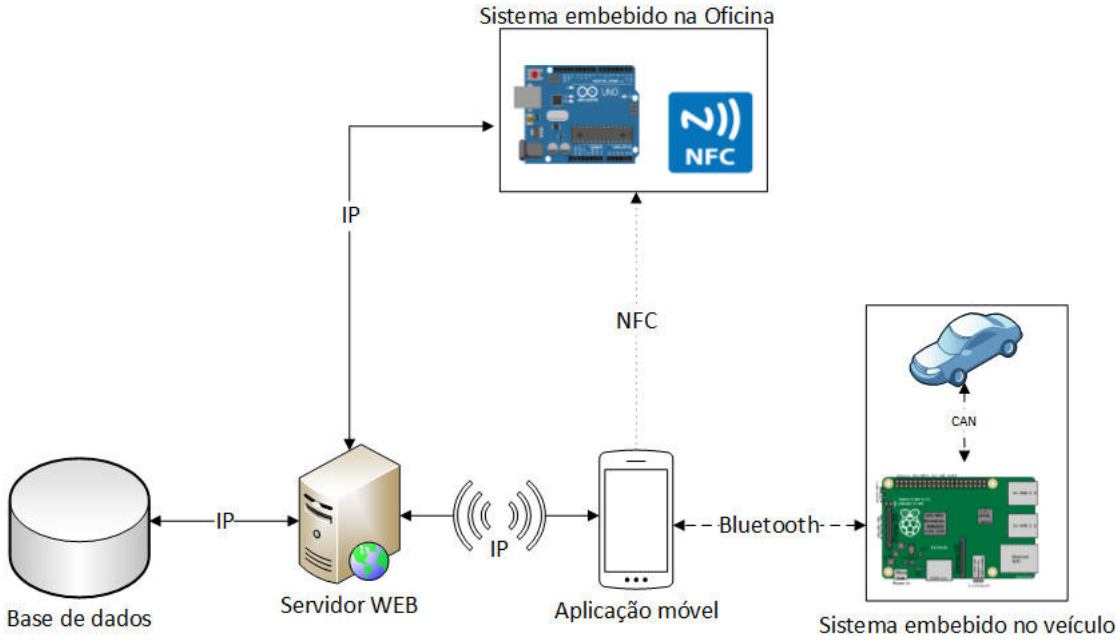


Figura 5.2 – Diagrama de arquitetura OwnGarage (com sistema embebido na oficina).

Nesta nova arquitetura, a adição dum sistema embebido na oficina, permite que quando um cliente se deslocar à oficina, na abertura duma nova ficha de reparação possa ser utilizado o dispositivo móvel (caso a aplicação móvel esteja instalada) para identificação do cliente e do automóvel. Este sistema embebido na oficina é composto por uma Arduino e por um leitor *Near Field Communication* (NFC), e pode agilizar o registo de novas reparações, trazendo assim mais comodidade. Para que este novo sistema embebido seja utilizado, é necessário proceder à alteração da aplicação móvel e da aplicação web, e o dispositivo móvel terá de possuir leitor NFC.

BIBLIOGRAFIA

AA1Car. (s.d.). *Manifold Absolute Pressure MAP Sensors*. Obtido de https://www.aa1car.com/library/map_sensors.htm

Abdulaziz Alshammari, M. A. (Setembro de 2018). *Classification Approach for Intrusion Detection in Vehicle Systems* . Obtido de https://www.researchgate.net/publication/328607559_Classification_Approach_for_Intrusion_Detection_in_Vehicle_Systems

Ahmad Aljaafreh, M. K.-F.-S.-E. (2011). Vehicular Data Acquisition System for Fleet. *Proceedings of 2011 IEEE International Conference on Vehicular Electronics and Safety*.

Amir Khajepour, M. S. (2014). *Electric and Hybrid Vehicles: Technologies, Modeling and Control - A Mechatronic Approach.*

AutoTap. (2013). Obtido de The OBDII Home Page: http://www.obdii.com

BOSCH. (s.d.). *Can Specification.* Obtido de http://esd.cs.ucr.edu/webres/can20.pdf

Dostál, R. (23 de Outubro de 2010). *Android Bluetooth Chat - client in Python*. Obtido de Radek Dostál 4 you: http://www.radekdostal.com/content/android-bluetooth-chat-client-python

Dwernychuk, J. (17 de Setembro de 2017). *SHA256 Encryption with Python*. Obtido de Medium: https://medium.com/@dwernychukjosh/sha256-encryption-with-python-bf216db497f9

Github. (2017). *An example using Python3 and AES criptography*. Obtido de https://gist.github.com/gustavohenrique/79cc95cc351d975a075f18a5c9f49319

Ilya Kolmanovsky, K. M. (2011). Estimation of fuel flow for telematics-enabled adaptive fuel and time efficient vehicle routing. *11th International Conference on ITS Telecommunications*.

Khazi, R. (18 de Junho de 2017). *Controller area network (CAN bus)*. Obtido de https://www.slideshare.net/raziuddinkhazi/controller-area-network-can-bus-ppt

Kristian Smith, J. M. (2013). OBDII Data Logger Design for Large-Scale Deployments. *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*.

Malintha Amarasinghe, S. K. (2015). Cloud-Based Driver Monitoring and Vehicle Diagnostic with OBD2 Telematics. *International Conference on Electro/Information Technology (EIT)*.

Mario Farrugia, J. P. (2016). The Usefulness of Diesel Vehicle Onboard Diagnostics (OBD) Information. *17th International Conference on Mechatronics - Mechatronika (ME)*.

MatthesRieke. (21 de Janeiro de 2016). *enviroCar*. Obtido de 52°North Wiki: https://wiki.52north.org/Projects/EnviroCar

Monroe, J. B. (Janeiro de 2013). *Overview of 3.3V CAN (Controller Area Network) Transceivers.* Obtido de http://www.ti.com/lit/an/slla337/slla337.pdf

Moore, A. (18 de Março de 2013). *THE 10 MODES OF OBDII*. Obtido de SearchAutoParts: https://www.searchautoparts.com/motorage/technicians/drivability/10-modes-obdii?page=0,1

Oracle Corporation. (2019). *MySQL :: MySQL 8.0 Reference Manual :: 24.4.2 Event Scheduler Configuration*. Obtido de https://dev.mysql.com/doc/refman/8.0/en/events-configuration.html

Parikh, B. (2016). *CAN Protocol - Understanding the Controller Area Network Protocol*. Obtido de EngineersGarage: https://www.engineersgarage.com/article/what-is-controller-area-network?page=1

Physics Forums. (21 de Fevereiro de 2017). *Air fuel ratio effects in diesel vs. petrol engines*. Obtido de Physics Forums: https://www.physicsforums.com/threads/air-fuel-ratio-effects-in-diesel-vs-petrol-engines.904965/

Samarins. (15 de Dezembro de 2018). *Mass Air flow Sensor (MAF): how it works, symptoms, problems, testing*. Obtido de https://www.samarins.com/glossary/airflow_sensor.html

Stack Overflow. (18 de Junho de 2010). *MySQL Event Scheduler on a specific time everyday*. Obtido de Stack Overflow: https://stackoverflow.com/questions/3070277/mysql-event-scheduler-on-a-specific-time-everyday

Stack overflow. (22 de Julho de 2011). *android encryption/decryption with AES*. Obtido de Stack overflow: https://stackoverflow.com/questions/6788018/android-encryption-decryption-with-aes

Stark, A. (s.d.). *Air-fuel ratio, lambda and engine performance*. Obtido de https://x-engineer.org/automotive-engineering/internal-combustion-engines/performance/air-fuel-ratio-lambda-engine-performance/

Wikipedia. (7 de Junho de 2017). *OBD-II PIDs*. Obtido de https://en.wikipedia.org/wiki/OBD-II_PIDs