UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

APLICAÇÃO PARA MONITORAMENTO VEICULAR EM TEMPO REAL

MAICON MACHADO GERARDI DA SILVA

BLUMENAU
2017

MAICON MACHADO GERARDI DA SILVA

APLICAÇÃO PARA MONITORAMENTO VEICULAR EM TEMPO REAL

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Miguel Alexandre Wisintainer - Orientador

BLUMENAU
2017

APLICAÇÃO PARA MONITORAMENTO VEICULAR EM TEMPO REAL

Por

MAICON MACHADO GERARDI DA SILVA

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof(a). Nome do(a) Professor(a), Titulação – Orientador, FURB
Membro:	Prof(a). Nome do(a) Professor(a), Titulação – FURB
Membro:	Prof(a). Nome do(a) Professor(a), Titulação – FURB

Blumenau, dia de mês de ano [data da apresentação]

Dedico este trabalho à minha família, amigos e colegas que me apoiaram e ajudaram para a realização do mesmo.

AGRADECIMENTOS

Primeiramente agradeço à Deus por prover conhecimento, saúde, paciência e capacidade para a realização deste trabalho.

À minha família, sem eles nada disso seria possível. Em especial minha mãe Maristela Machado Gerardi e meu pai Eri Junior Miranda Lopes por financiar boa parte do meu sonho que é tornar-me um bacharel em ciência da computação. À minha irmã Milena Kohhausch, que ajudou a corrigir e auxiliou na ortografia deste documento e do pré-projeto.

À minha namorada Jaine Marcírio, por me apoiar, estar ao meu lado e ter paciência comigo nessa etapa da minha vida.

Ao professor Miguel Alexandre Wisintainer pela orientação, entusiasmo e perseverança principalmente nos momentos iniciais, onde dedicou tempo para viabilizar este trabalho. Agradeço por acreditar no projeto, na minha capacidade e por me orientar de forma adequada para a realização e sucesso deste projeto.

Ao Charles Trevizan e Claumir dos Santos da empresa Dynamix Software, por flexibilizar o meu horário de trabalho para desenvolvolver este projeto e o apoio e financeiro com a metade da mensalidade da faculdade.

Ao Nykolas Baumgarten por emprestar o seu modem 4G; agradeço também por me inspirar com seu excelente trabalho de conclusão de curso, bem como Ricardo Starosky.

Aos meus colegas de trabalho e amigos Silvio Gonçalves Neto e Alessandro Jefferson Carvalho por me darem ideias de melhoria do projeto, aconselharem com o desenvolvimento e emprestarem seus veículos para testes da aplicação.

Ao meu amigo Renan Ramos dos Santos Vieira por emprestar o carro para testes e também me ajudar com o seu conhecimento sobre infraestrutura de redes.

Ao meu amigo e colega Plamedi L. Lusembo por ajudar no decorrer do trabalho com o chip da operadora TIM, além de apoios com a monografia e o desenvolvimento.

Ao professor Francisco Adell Péricas por me auxiliar com explicações e orientações sobre infraestrutura de redes 3G e servidores virtuais utilizados na comunicação do servidor deste projeto.

À todas as pessoas que contribuíram direta e indiretamente para a realização deste trabalho.

“ Nada se cria, tudo se transforma ”.

Antonie Lavoisier

RESUMO

O resumo é uma apresentação concisa dos pontos relevantes de um texto. Informa suficientemente ao leitor, para que este possa decidir sobre a conveniência da leitura do texto inteiro. Deve conter OBRIGATORIAMENTE o **OBJETIVO**, **METODOLOGIA**, **RESULTADOS** e **CONCLUSÕES**. O resumo deve conter de 150 a 500 palavras e deve ser composto de uma sequência corrente de frases concisas e não de uma enumeração de tópicos. O resumo deve ser escrito em um único texto corrido (sem parágrafos). Deve-se usar a terceira pessoa do singular e verbo na voz ativa (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2003).

Palavras-chave: Monitoramento veicular. IOT. Internet das coisas. EML327. GPS. Geolocalização. OnBoard Diagnostic. OBD. OBDII. OBD2. Raspberry Pi. DTC. Diagnostic Trouble Code.

[Palavras-chave são separadas por ponto, com a primeira letra maiúscula. Caso uma palavra-chave seja composta por mais de uma palavra, somente a primeira deve ser escrita com letra maiúscula, sendo que as demais iniciam com letra minúscula, desde que não sejam nomes próprios.]

ABSTRACT

Abstract é o resumo traduzido para o inglês. *Abstract* vem em uma nova folha, logo após o resumo. Escrever com letra normal (sem itálico).

Key-words: Computer science. Monograph. Abstract. Format.

[*Key-words* são separadas por ponto, com a primeira letra maiúscula. Caso uma *key-word* seja composta por mais de uma palavra, somente a primeira deve ser escrita com letra maiúscula, sendo que as demais iniciam com letra minúscula, desde que não sejam nomes próprios.]

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

2G - 2rd Generation

3G - 3rd Generation

4G - 4rd Generation

CAN - Controller Area Network

CI - Cicuinto Integrado

CPU - Central Processing Unit

CSI - Camera Serial Interface

DDNS - Dynamic Domain Name System

DTC - Diagnostic Trouble Code

ECU - Engine Control Unit

GB - Gigabyte

Ghz - Giga-hertz

GND - Ground

GPIO - General Purpose Input/Output

GPS – Global Position System

HDMI - High-Definition Multimedia Interface

IOT - Internet of Things

IP - Internet Protocol

ISO - International Starndardization Organization

JSON - JavaScript Object Notation

LED - Light Emitting Diode

MB - Megabyte

MB - Megabytes

Mhz - Mega-hertz

MIL - Malfunction Indicator Lamp

OBD - On-Board Diagnostic

OBD2 - On-Board Diagnostic 2

PHP - Personal Home Pages

PID - Códigos de Parâmetros

RAM - Random Access Memory

RF - Requisito Funcional

RNF - Requisito não Funcional

SAE - Society for Automotive Engineers

SD - SanDisk

SMS - Short Message Service

SO - Sistema operacional

TCP/IP - Transmission Control Protocol/Internet Protocol

UART - Universal Asynchronous Receiver/Transmitter

USB - Universal Serial Bus

VIN - Vehicle Identification Number

SUMÁRIO

‘

1 INTRODUÇÃO

Nos sete primeiros meses de 2016 foram furtados em Santa Catarina 3.164 carros e pick-ups nacionais e importados com seguro, segundo Odega (2016). Conforme os dados da Superintendência de Seguros Privados (SUSEP), só no primeiro semestre de 2016 as seguradoras registraram 118 mil veículos roubados/furtados no Brasil (ODEGA, 2016).

Odega (2016) cita os veículos com mais índices de roubo, são eles:

a) Chevrolet Celta 1.0 com 6.055 ocorrências de um total de 170.524 veículos segurados;
b) VW Volkswagen Gol 1.0 com 4.514 ocorrências de 253.594 veículos com seguro;
c) Fiat Palio 1.0 com 4.127 ocorrências de um total de 219.654 com seguro.

No ano de 2017, o número de roubos a veículos aumentou 20% em Ribeirão Preto no estado de São Paulo. Segundo G1 Ribeirão e Franca (2017, p. 1), "[...] 60 roubos de carros e motocicletas ocorreram durante janeiro de 2017. Ao todo, 50 casos ocorreram durante os primeiros trinta dias de 2016 [...]".

Além dos furtos, ainda pode-se analisar a quantidade de veículos com falhas nas estradas. Entre janeiro e outubro de 2014 foram registrados pouco mais de meio milhão de veículos que ficaram parados nos mais de 6 mil quilômetros de rodovias do Programa de Concessões Rodoviárias do Estado de São Paulo por apresentarem problemas de manutenção, por exemplo pneu furado e superaquecimento do motor (SOUZA, 2016). Essa estatística equivale à um pouco mais de 83 carros parados por quilômetro nesse período. Uma pesquisa realizada pelo Instituto Scaringella de Trânsito aponta que a falta de manutenção preventiva no automóvel é relacionada com 30% dos acidentes rodoviários e urbanos no Brasil (CZERWONKA, 2016). Ainda segundo o autor, a manutenção preventiva do veículo não só beneficia a segurança no trânsito, bem como ajuda o condutor a economizar. Cuidar do carro antes que alguma peça apresente defeito custa, em média 30% a menos do que fazer somente a checagem de rotina.

Diante desse cenário, Baumgarten (2016) desenvolveu um dispositivo que possibilitasse o rastreamento veicular através de geolocalização e uma imagem capturada através de uma câmera acoplada neste dispositivo. Paralelamente à Baumgarten (2016), Staroski (2016) desenvolveu um protótipo de software embarcado em uma placa Raspberry Pi para capturar dados da porta On-Board Diagnostic (OBD) de um veículos e disponibilizá-los em uma página *web*.

Com base nesses argumentos, este trabalho consiste em integrar as principais funcionalidades desenvolvidas por Baumgarten (2016) e o protótipo de Starosky (2016) em uma única plataforma, isto é realizado através de um software embarcado em uma placa Raspberry Pi Zero W para capturar a geolocalização de um veículo, imagens deste automóvel e os dados de sua porta OBD. Foi desenvolvida uma aplicação *mobile* para disponibilizar os dados do software embarcado.

1.1 OBJETIVOS

O objetivo deste trabalho é a construção de uma aplicação que abrange o desenvolvimento de um software embarcado em uma placa Raspberry Pi Zero W para coletar a geolocalização, imagens de uma câmera e dados da porta OBD de um automóvel, bem como o desenvolvimento de uma aplicação *mobile* para capturar as informações desse software embarcado.

Os objetivos específicos são:

a) integrar a placa Raspberry Pi Zero W com um módulo Global Positioning System (GPS), um módulo Bluetooth OBD e uma câmera;
b) desenvolver um software embarcado onde será possível verificar a localização atual, as últimas localizações do veículo, capturar imagens e disponibilizar informações da porta OBD;
c) desenvolver uma aplicação *mobile* para consultar as informações disponíveis pelo software embarcado;
d) notificar o usuário sobre falhas no motor retornados pela porta OBD.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem como objetivo abordar os principais assuntos utilizados para a realização deste trabalho. Eles foram subdivididos em cinco partes, onde a seção 2.1 apresenta a Internet das Coisas (Internet of Things - IOT). A seção 2.2 expõe os assuntos relacionados a OBD:

a) seção 2.2.1: OBD versão 1;
b) seção 2.2.2: OBD versão 2;
c) seção 2.2.3: adaptadores ELM327 utilizados na leitura da porta OBD versão 2;
d) seção 2.2.4: protocolos de comunicação;
e) seção 2.2.5: serviços de diagnósticos OBD versão 2;
f) seção 2.2.6: por fim, é explicado sobre códigos de erro (Diagnostic Trouble Code – DTC).

Após os assuntos explorados sobre OBD, é apresentado na seção 2.3 a plataforma Raspberry Pi. Na seção 2.4 são descritos dois trabalhos correlatos e, por fim, a seção 2.5 são apresentadas duas ferramentas atuais.

2.1 INTERNET DAS COISAS

Em meados de 1991 iniciou-se a discussão sobre a conexão de objetos na rede. Nesse período, a conexão Transmission Control Protocol/Internet Protocol (TCP/IP) e a Internet se tornam acessíveis. Com isso, Bill Joy, o cofundador da Sun Microsystems foi o principal pensador da ideia de conectar várias redes de dispositivos. Por volta de 1999, Kevin Aston do Massachusets Institute Technology (MIT) propôs o termo Internet das Coisas ou Internet of Things (IOT) após dez anos de estudos e realização de projetos, ele escreveu um artigo chamado “A Coisa da Internet das Coisas” para o RFID Journal. De acordo com Aston, a falta de tempo é um fator que contribui para a necessidade de conectar os dispositivos à Internet (APLICAÇÕES DE AUTOMAÇÃO EM IOT - INTERNET OF THINGS, 2016, p. 3).

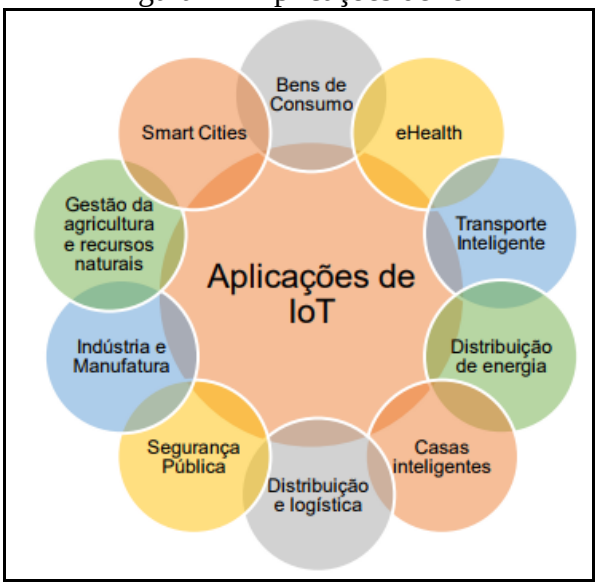
Santos et al. (2016) afirma que a IOT é uma extensão da Internet atual que proporciona para quaisquer tarefas do dia-a-dia, uma maior capacidade computacional e de comunicação por meio da Internet. A conexão com a rede mundial de computadores viabiliza controlar os objetos remotamente e com isso, gerar oportunidades no âmbito acadêmico e industrial. Para Aplicações De Automação Em Iot - Internet Of Things (2016), IOT é um conceito que surgiu com a convergência de tecnologias que envolvem comunicação sem fio, sistemas embarcados e eletromecânicos. Com isso, os principais componentes da rede IOT são:

a) as próprias coisas, por exemplo: aparelhos eletrônicos, sensores, atuadores, computadores e celulares;
b) redes de comunicação.

As aplicações de IOT são diversas, incluem desde tarefas diárias até a sociedade como um todo (MANCINI, 2017). Dias (2016) afirma que o conceito de IOT transforma o mundo em um *smart world*. Na Figura 1 são ilustradas as diversas aplicações da Internet das Coisas e dentro deste cenário, é explicado abaixo cada conceito:

a) bens de consumo: adquiridos pelos consumidores, como *smart* TV e smartphone;
b) eHealth: boa forma, bioeletrônica e cuidados com a saúde;
c) transporte inteligente: notificação das condições de tráfego, controle inteligente de rotas e monitoramento remoto do veículo;
d) distribuição de energia: acompanhamento de instalações elétricas e subestações inteligentes;
e) casas inteligentes: medições remotas de consumo, economia de energia e controle de equipamentos remotamente;
f) distribuição e logística: *smart e-commerce*, rastreabilidade, gerenciamento na distribuição e inventário;
g) segurança pública: monitoramento no transporte de cargas perigosas, monitoramento de construções e utilidades públicas;
h) indústria e manufatura: economia de energia, controle de poluição, segurança na manufatura, ciclo de vida dos produtos, rastreamento e cadeia de abastecimento;
i) gestão da agricultura e dos recursos naturais: segurança e rastreio de produtos agrícolas, monitoramento ambiental para produção, cultivo e gerenciamento no processo de produção;
j) *smart cities*: monitoramento estrutural, como por exemplo vibrações e condições dos materiais em edifícios, pontes e monumentos históricos, iluminação inteligente, monitoramento para prevenção de incêndios, estradas com avisos de desvio conforme condições climáticas, monitoramento em tempo real de espaços de estacionamento e auxílio na coleta de lixo.

Figura 1 – Aplicações de IoT



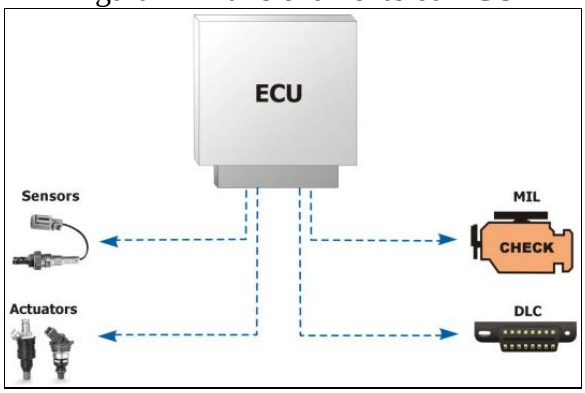
Fonte: Mancini (2017, p. 6).

A IOT é um grande desafio em seu nível conceitual e tecnológico, pois são várias tecnologias embutidas num único sistema, além de ter muitos segmentos de mercado com aplicação de internet das coisas e cada vez surgem mais aplicações (DIAS, 2016). Segundo Mancini (2016), a Internet das Coisas fará parte da estratégia e gestão, diante disso, serão necessários novos modelos empresarias. Ainda segundo a autora, abrem-se assim novas oportunidades para a atuação de gerentes de projetos para implementar essas soluções e lidar com a transformação organizacional, tecnológica e sócio cultural.

2.2 OBD

Segundo Santos (2016), On-Board Diagnostic (OBD) é um sistema de autodiagnóstico disponível na maioria dos veículos. A ligação ao sistema ocorre por meio de um conector padronizado, que foi sancionado como obrigatório na Europa e nos Estados Unidos a partir de 1996. Um sistema OBD básico consiste em uma Unidade de Controle Eletrônico (Engine Control Unit - ECU) que utiliza a entrada de vários sensores para manipular atuadores, como por exemplo sensores de oxigênio controlando injetores de combustível, esse fim é utilizado para obter um desempenho desejado (OBD SOLUTIONS, 2017). Ainda segundo o autor, a luz conhecida como Malfunction Indicator Lamp (Luz de Mal Funcionamento - MIL) fornece aviso prévio de avaria no veículo ao proprietário. Um veículo moderno suporta centenas de parâmetros que podem ser acessados através do Diagnostic Link Connector (DLC) usando um dispositivo chamado ferramenta de verificação. Na Figura 2 podemos visualizar um diagrama do funcionamento de uma ECU.

Figura 2 – Funcionamento da ECU



Fonte: Obd Solutions (2017, p.1).

Segundo Santos (2016), no Brasil foi sancionado como obrigatório somente a partir de 2010, com o padrão da segunda geração OBD. O autor cita que “A medida tem a finalidade de fiscalizar a emissão de gases poluentes na atmosfera, dado que, alguns países possuem acordos mundiais em que se comprometem com a preservação ambiental, como o protocolo de Kyoto.”.

2.2.1 OBD1

O sistema OBD1 teve pouco êxito por causa da falta de padronização entre os fabricantes de veículos (como pode ser observado na Figura 3) e pela falta de informações específicas em cada sistema. As dificuldades técnicas de se obter as informações corretas de todos os tipos de veículos inviabilizaram o plano de inspeções veiculares (MCCORD, 2011, p.1).

Segundo Machado e Oliveira (2007), apesar desses fatores de falta de padronização, os sistemas apresentavam os seguintes itens:

a) sensor de oxigênio;
b) sistema de EGR;
c) sistema de combustível;
d) componentes eletrônicos;
e) sistemas eletrônicos;
f) informação de diagnóstico;
g) códigos de erros.

Figura 3 - Exemplo de falta de padronização de conectores OBD-I

Fonte: MCCORD (2011, p.1).

2.2.2 OBD2

No início dos anos 90, a Society of Automotive Engineers (SAE) e a International Standardization Organization (ISO) emitiram um conjunto de normas que descrevem o intercâmbio de informações entre ECUs e uma ferramenta de diagnóstico (OBD SOLUTIONS, 2017). Ainda segundo o autor, todos os veículos compatíveis com On-Board Diagnostic 2 (OBD2) foram obrigados a utilizar um conector de diagnóstico padrão (SAE J1962) e comunicar através de um protocolo padrão.

A diferença da atualização para o OBD2 foi a eliminação do grande defeito do OBD1 que consiste na falta de coerência entre os vários sistemas existentes (MACHADO; OLIVEIRA, 2007). Segundo Machado e Oliveira (2007), ouve uma normalização de procedimentos, ou seja, uma estandardização no que diz respeito a métodos de conexão e acima de tudo a nível de protocolos. A lista de itens disponíveis para acesso e controle também foi ampliada:

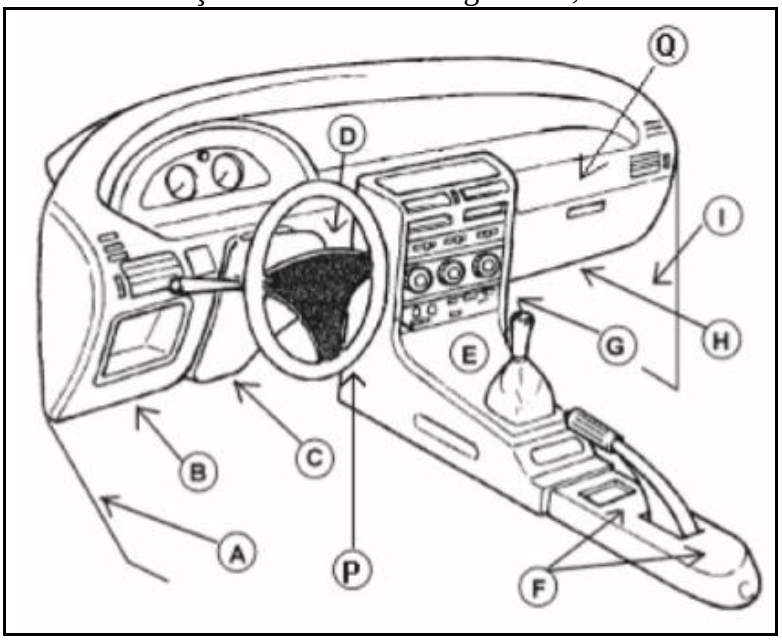
a) sensor de oxigênio;
b) sistema de EGR;
c) sistema de combustível;
d) componentes eletrônicos;
e) sistemas eletrônicos;
f) eficiência de catalisador;

g) aquecimento de catalisador;
h) combustão espontânea;
i) sistema de evaporação;
j) sistema de ar secundário;
k) informações de diagnóstico;
l) códigos de falha;
m) parâmetros do motor;
n) memorização de avarias;
o) estandardização de ligações.

Machado e Oliveira (2007) afirmam que com esses itens a ser constantemente analisados, conseguiu-se cada vez mais diminuir a emissão de gases poluentes. Além disso, o conector OBD2 (ilustrado na Figura 5) está localizado perto do console central do carro na maioria dos casos. Na Figura 4, são apresentadas as possíveis localizações da tomada de diagnóstico. Ela deve atender as seguintes especificações:

a) próximo ao assento do passageiro ou motorista;
b) próximo ao painel de instrumentos;
c) distância de 300mm além da ECU;
d) fácil acesso ao assento do motorista;
e) entre a coluna de direção e a ECU.

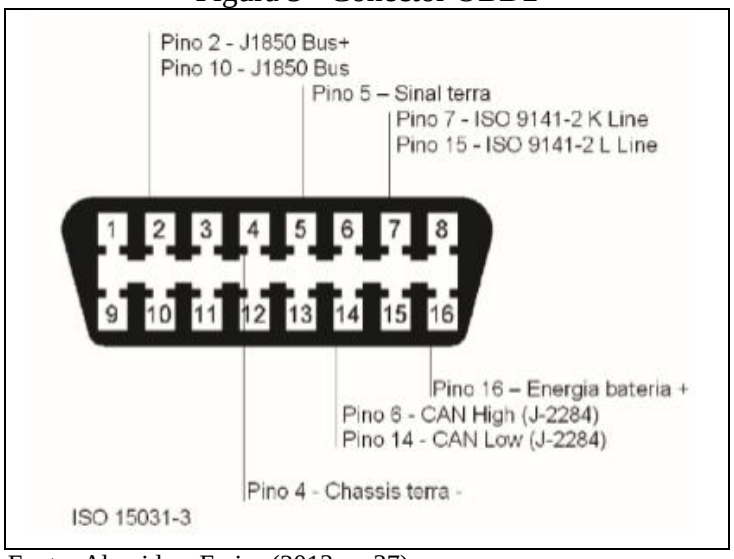
Figura 4 - Localização da tomada de diagnóstico, interior do veículo.



Fonte: Almeida e Farias (2013 p. 36)

Segundo Almeida e Farias (2013), o conector OBD2 possui 16 pinos. Esses, juntamente com seus protocolos de comunicação são ilustrados na Figura 5. Pacheco (2016) afirma que este conector precisa estar facilmente acessível à partir do banco do motorista, caso haja alguma tampa ou conector, o mesmo deve ser de fácil remoção sem o uso de ferramentas.

Figura 5 - Conector OBD2



Fonte: Almeida e Farias (2013, p. 37)

2.2.3 Adaptadores ELM237

Os adaptadores ELM327 são dispositivos utilizados para acessar os dados da porta OBD2 de um veículo (OUTILS OBD FACILE, 2017, p. 1). Eles possuem um circuito integrado (CI) chamado ELM327 que são fabricados pela empresa Elm Electronics (ELM ELECTRONICS, 2017, p. 1). Segundo Almeida e Farias (2013), este CI é baseado em outros, como o ELM320, ELM322 e ELM323. Conforme Figura 6, existem diferentes adaptadores, são eles:

a) RS232 (RS ou Série): é utilizado para comunicação serial entre computador e interface OBD2 do veículo. Segundo Outils Obd Facile (2017), este tipo de saída está depreciando-se de computadores modernos;
b) Universal Serial Bus (USB);
c) Bluetooth;
d) *wireless*.

Figura 6 - Diferentes adaptadores ELM 327

Fonte: Santos (2016, p. 26).

Segundo Elm Electronics (2017), na maioria dos veículos são utilizados os protocolos Controller Area Network (CAN) (ISO 15765-4), porém, o ELM327 foi projetado para suportar todos os protocolos OBD2 padrão. Por ser multiprotocolo, o autor afirma que é o CI ELM327 é atualizado desde 2005 e possui quatro versões, que estão ilustradas no Quadro 1.

Quadro 1 - Diferentes versões ELM327

	ELM327 v1.3a	ELM327 v1.4b	ELM327 v2.2	ELM327L v2.2
Tensão operacional	4.5V a 5.5V	4.5V a 5.5V	4.2V a 5.5V	2,0 V a 5,5 V
Modo Low Power (Sleep)	Não	sim	sim	sim
Configurações Retained on Wake	-	Não	sim	sim
RS232 Transmit Buffer Bytes	256	256	512	2048
Comandos AT	93	115	128	128
Verificação de frequência da CAN	Não	Não	sim	sim
Suporte de resposta pendente (7F)	Não	Não	sim	sim

Fonte: Elm Electronics (2017, p. 1).

Segundo Almeida e Farias (2013), para tornar o circuito integrado funcional, é necessária uma configuração prévia dependendo do dispositivo utilizado, por exemplo, se utilizar o adaptador Bluetooth, requer pareá-lo com o sistema operacional (SO). Após a

configuração, é possível utilizar comunicação serial através de um computador ou smartphone, no caso do sistema operacional Windows, podem-se utilizar softwares como exemplo: Putty, ZTerm e Tera Term. No sistema operacional Android podem-se utilizar aplicativos, como por exemplo: Serial USB Terminal e UsbTerminal.

Almeida e Farias (2013) afirmam que após estabelecer uma conexão serial com o adaptador, o ELM327 envia uma mensagem informando a versão do circuito integrado. Além disso, ele envia um caractere “>” indicando que está pronto para receber um comando. Ainda, segundo os autores, os comandos podem ser de dois tipos:

a) comandos internos: são destinados para utilização interna do circuito ELM327 e começam com os caracteres “AT”;
b) comandos para barramento: utilizados no barramento OBD2 e contém apenas códigos ASCII para dígitos hexadecimais (de 0 a F).

As mensagem que não são compreendidas pelo ELM327 respondem com um ponto de interrogação “?”. Porém, segundo Almeida e Farias (2013), isso não significa que a mensagem foi desentendida, pode ser que a mensagem não é suportada pelo circuito. Além disso, caracteres maiúsculos e minúsculos não são distinguidos, bem como são ignorados caracteres de espaços e todos os caracteres de controle (tab, enter, etc).

2.2.4 Protocolos de comunicação

Cada protocolo de comunicação possui características específicas (tempo de resposta, banda, redundância, detecção de erros, arquitetura de redes e software de programação), sendo normal encontrar mais de um barramento implantado em um veículo (ALMEIDA; FARIAS, 2013, p. 51). Com isso, os autores afirmam que, em 1994 a Sociedade dos Engenheiros Automotivos dos Estados Unidos (Society for Automotive Engineers - SAE) definiu uma classificação para os protocolos de comunicação automotiva. Esta classificação é baseada na velocidade de transmissão de dados e a função que são distribuídas pela rede.

Segundo Bastos (2012), o conector OBD2 disponibiliza cinco protocolos automotivos, são eles:

a) SAE J1850 PWM: possui a taxa de transferência de 41.6 Kbps, utiliza 2 pinos do conector OBD2 e o tamanho da sua mensagem é de 12 bytes (padrão utilizado pela Ford Motors);
b) SAE J1850 VPW: possui um único fio de conexão, por isso é considerado de baixo custo. Taxa de transferência de 10.4 Kbps e tamanho para mensagens de 12 bytes (utilizado pela General Motors (GM) com o nome de GM Class 2);

c) ISO 1941-2: a comunicação é assíncrona Universal Asynchronous Receiver/Transmitter (UART), taxa de transmissão 10.4 Kbps e mensagem de 5 a 11 bytes (protocolo utilizado pela Crysler, fabricantes europeus e asiáticos);
d) ISO 14230: popularmente chamado de Keyword 2000 (KW2000), é apenas um link de diagnóstico e não pode ser utilizado para transmitir mensagens. Foi bastante utilizado antes mesmo da OBD2 por fabricantes como Bosh, Opel e outros fabricantes europeus, que pressionavam órgãos regulamentadores para viabilizar a utilização deste link. Sua liberação ocorreu na década de 90, e só foi aprovada por ter o protocolo e os requerimentos de *hardware* quase idênticos ao ISO 9141-2. Este protocolo possui a velocidade entre 1.2 e 10.4 Kbps e suas mensagens podem conter até 255 bytes;
e) ISO 15765: mais conhecido como Controller Area Network (CAN), foi desenvolvido pela Bosh, sua comunicação é serial e sua velocidade é de 500 Kbps. O padrão ISO 15765-4 determina os requisitos para a aplicação OBD.

2.2.5 Serviços de diagnóstico

Os serviços de diagnóstico de um sistema OBD2 são organizados por modos de operação e códigos de parâmetros (PIDs) (SANTOS, 2016, p.19). O padrão OBD2 regulamenta um conjunto de PIDs e modos de operação, sendo que alguns deles são de implementação obrigatória pelo fabricante, especialmente os que são relacionados à emissão de gases poluentes. Alguns PIDs possuem implementação opcional que o fabricante opta ou não por disponibilizar de acordo com a sua necessidade.

Segundo Almeida e Farias (2013), existem dez serviços disponíveis, de modo que cada veículo ou ECU deverá implementar seus serviços de acordo com a sua legislação. Santos (2016) afirma que cada serviço é um valor hexadecimal de dois dígitos, que deve estar entre 0x01 e 0x0A. Cada operação pode ter até 256 PIDs, que também são representados por um número hexadecimal de dois dígitos.

Existe também o serviço 0x00, que é reservado à aderência do veículo ao padrão OBD, ou seja, a resposta retornada por ele é um vetor de bits em que cada valor binário indica se o PID correspondente é suportado ou não pela ECU (SANTOS, 2016, p. 19). Os serviços 0x01 à 0x0A são descritos abaixo:

a) serviço 0x01: Bastos (2012) relata que este serviço permite o acesso às informações de dados relacionados ao *powertrain*, dentre eles sinais de entrada/saída analógicas e digitais, além de informações do sistema. Alguns

exemplos de PIDs suportados por este serviço são:

- PID 05: temperatura do fluido de arrefecimento (°C),
- PID 00: velocidade do veículo (km/h),
- PID 11: posição da borboleta;

b) serviço 0x02: exibe dados do *Freeze Frame*, ou seja, no momento que aconteceu uma falha estes dados são "congelados" e armazenados neste serviço. O PID 02 deste serviço indica o código de erro (Diagnostic Trouble Code - DTC) que causou o *Freeze Frame*;

c) serviço 0x03: lista os DTCs confirmados que impactam emissões, este serviço permite à ferramenta de *scanner* listar todos os DTCs de cinco dígitos que estão presentes no momento ou que já iluminaram a lâmpada de mal funcionamento (MIL) recentemente;

d) serviço 0x04: segundo Bastos (2012), este serviço permite que a ferramenta de *scanner* possa comandar a ECU para limpar as informações de diagnóstico, incluindo os DTCs armazenados, dados de *Freeze Frame* e distância que ocorreu a falha. Este serviço também reinicia todos os contadores de diagnóstico e retira as ações de degradação do sistema. Ele só deverá funcionar com a ignição ligada e motor desligado, sendo aplicado a todos os controladores ECUs;

e) serviço 0x05: requisita resultados de teste de sensor de oxigênio, este serviço não é suportado pela CAN e sua funcionalidade está implementada no serviço 0x06;

f) serviço 0x06: Bastos (2012) afirma que este serviço informa testes de monitoramento para componentes e sistemas específicos que são constantemente monitorados (ex: *misfire* ou falha de combustão). O resultado é exibir o último valor vigente do teste e também os limites máximo e mínimo;

g) serviço 0x07: corresponde aos códigos DTCs pendentes de acender a lâmpada de mal funcionamento MIL no ciclo de condução vigente e anterior. Segundo Bastos (2012), este serviço é independente do serviço 0x03, porém, com o mesmo formato. O seu principal objetivo é auxiliar o técnico a verificar problemas em componentes durante o reparo dos mesmos após suas substituições, bem como na limpeza das informações de diagnóstico com o serviço 0x04.

h) serviço 0x08: Segundo Santos (2016), este serviço é bidirecional, ou seja, com ele é possível ler, alterar parâmetros de operação e também é possível testar diferentes hipóteses para identificar a causa do problema. Este serviço deve ser utilizado com cuidado, pois pode causar avarias ao veículo. Santos (2016) também afirma que,

através desse módulo, é possível alterar a potência do motor em alguns cavalos de força apenas por meio de reprogramação da ECU;

i) serviço 0x09: permite obter informações sobre o software instalado na ECU. É mais utilizado para consultar o Vehicle Identification Number (VIN), que é o código identificador único para carros e caminhões. Também serve para identificar a versão do software que está instalado e este serviço permite obter relatórios sobre sistema catalizador, sensor de oxigênio, vazamento no sistema de evaporação, controle de pressão, contadores de eventos como o número de ignição e testes de autodiagnóstico (SANTOS, 2016, p. 21);

j) serviço 0x0A: por fim, este módulo ou serviço refere-se à lista de todos os erros gerados cada vez que a luz MIL foi acesa ou uma indicação de falha foi ativada. Diferente do serviço 0x03 e 0x07, ele não pode ser apagado nem mesmo com a desconexão da alimentação da ECU. Isto se dá pelo fato de manter um histórico de anomalias do veículo e evita que o condutor ou técnico apague estes erros.

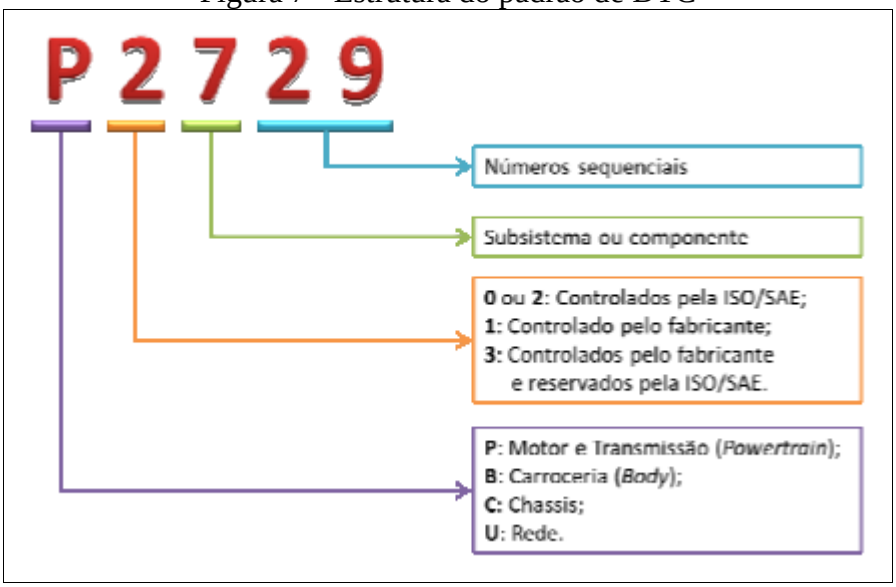
2.2.6 Diagnostic Trouble Code (DTC)

Diagnostic Trouble Code (DTC) são os códigos de erros retornados e armazenados pelo sistema OBD2 quando ocorre algum problema no ECU. Segundo Almeida e Farias (2013), para identificar falhas no sistema OBD2, são executados testes no circuito de todos os sensores e monitores procurando por possíveis curtos circuitos, circuitos abertos, plausibilidade do sinal e também é processada a ECU (codificação, memória e etc). Se uma falha for detectada por algum dos testes, o código de defeito (DTC) é armazenado e caso essa for relacionada com emissão de poluentes, a luz MIL será acionada. Almeida e Farias (2013) relatam que o DTC não é necessariamente a causa do defeito e sim, isolar a falha a uma área funcional específica do veículo.

Segundo Almeida e Farias (2013), para que o sistema de diagnose detecte a necessidade de gerar um código de falha, a informação deve ser pertinente por até 10 ciclos de motor. A fim de interpretar que esta falha foi sanada, é necessário que este código não esteja presente por no mínimo 40 ciclos de motor. Com intuito de apagar a MIL são necessários apenas 3 ciclos. Cada ciclo de motor é equivalente a ligá-lo, aquecê-lo, percorrer uma distância pré-determinada e verificar todos os sensores.

Os códigos de falhas DTC são formados por 2 bytes ou 16 bits e seguem um padrão explicados na Figura 7. Este padrão segue uma nomenclatura conforme norma SAE J2012 (BASTOS, 2012, p. 30).

Figura 7 - Estrutura do padrão de DTC



Fonte: Bastos (2012, p. 30)

Segundo Almeida e Farias (2013), o primeiro caractere (uma letra) refere-se ao sistema que a falha pertence (conforme Quadro 2).

Quadro 2 - Primeiro caractere do código de falha.

Letra	Valor binário	Significado
P	00	Motor *(Powertrain)*
C	01	Chassi *(Chassis)*
B	10	Corpo *(Body)*
U	11	Rede *(Network)*

Fonte: Almeida e Farias (2013, p. 38).

O primeiro dígito após o caractere (número de 0 a 3) é a entidade responsável pela criação do código. Através dele, é possível verificar se pertence à algum fabricante específico ou comum para todos os fabricantes (ALMEIDA; FARIAS, 2013). Os possíveis valores estão ilustrados no Quadro 3.

Quadro 3 - Primeiro valor numérico do código de falhas (segundo caractere)

VALOR	Entidade Responsável
0	ISO/SAE
1	Fabricante.
2	ISO/SAE
3	ISO/SAE, reservado e Fabricante

Fonte: Almeida e Farias (2013, p. 37).

O terceiro caractere refere-se a um subgrupo de funções do veículo. No Quadro 4, é apresentado os valores do terceiro dígito. O quarto e quinto dígito são falhas específicas do subgrupo identificado.

Quadro 4 - Terceiro dígito do código de falhas

Valor	Descrição
0	Sistema eletrônico completo.
1	Controle Ar / Combustível.
2	Controle Ar / Combustível; Circuito de injeção.
3	Sistema de ignição.
4	Controle Auxiliar de Emissões
5	Controle de velocidade do veículo e de rotação em marcha lenta.
6	Circuitos de entrada e saída da central eletrônica.
7	Transmissão.
8	Transmissão.

Fonte: Almeida e Farias (2013, p. 37).

Se a comunicação for feita através de protocolo serial, o valor do DTC é apresentado em hexadecimal e precisa ser decodificado para o código correspondente. O Quadro 5 ilustra um exemplo para a decodificação dó código C0123 para hexadecimal e binário.

Quadro 5 - Exemplo de decodificação DTC

Unidade	Valores															
Código DTC	C		0		1			2			3					
Binário	0	1	0	0	0	0	0	1	0	0	1	0	0	0	1	1
Hexadecimal	4				1			2			3					

Fonte: elaborado pelo autor.

2.3 RASPBERRY PI

Raspberry Pi é um computador do tamanho de um cartão de crédito que se conecta a um monitor ou uma TV, usa um teclado e mouse padrão e foi desenvolvido no Reino Unido pela Fundação Raspberry Pi. Todo o hardware é integrado à uma única placa, a Figura 8 demonstra a comparação de tamanho entre dois modelos de placas Raspberry Pi. O principal objetivo é promover o ensino da ciência da computação básica em escolas (RASPBERRY PI FOUNDATION, 2017, p. 1). Ainda segundo o autor, existem vários modelos disponíveis com as mais diversas configurações, o Quadro 6 cita suas principais diferenças e especificações entre elas.

Quadro 6 - Comparativo entre os modelos Raspberry Pi

Características / Modelos	Velocidade	RAM	Portas USB	Ethernet	Wireless e Bluetooth	Preço
Raspberry Pi Model A+	700Mhz	512MB	1	Não	Não	$20
Raspberry Pi Model B+	700Mhz	512MB	4	Sim	Não	$25
Raspberry Pi 2 Model B	900Mhz	1GB	4	Sim	Não	$35
Raspberry Pi 3 Model B	1200Mhz	1GB	4	Sim	Sim	$35
Raspberry Pi Zero	1000Mhz	512MB	1	Não	Não	$5
Raspberry Pi Zero W	1000Mhz	512MB	1	Não	Sim	$10

Fonte: adaptado de RASPBERRY PI FONDATION (2017, p. 1).

O modelo A+ é a variante de baixo custo, possuindo 512 Megabytes (MB) de Random Access Memory (RAM) e 700 Mega-hertz (Mhz) de processamento sem nenhuma conexão *wireless*, Bluetooth e Ethernet. Já a placa modelo B+ é a adaptação dela e por um preço razoável já vem com porta Ethernet (RASPBERRY PI FONDATION, 2017, p. 1).

Segundo Raspberry Pi Foundation (2017), em fevereiro de 2015 o modelo B+ foi substituído pelo Pi 2 Model B, que contém um processador *quad-core* ARM Cortex-A7 Central Processing Unit (CPU) de 900Mhz de processamento e 1 Gyga-byte (GB) de memória RAM. E o modelo Pi 3 Model B foi lançado em fevereiro de 2016, utilizando um processador ARM Cortex-A53 de 1.2 Giga-hertz (GHz), integrando *wireless*, Ethernet e Bluetooth versão 4.1. Na Figura 8, esta placa é a maior tamanho, este modelo é o mais utilizado e recomendado

para uso em escolas devido à sua flexibilidade para o aluno (RASPBERRY PI FONDATION, 2017, p. 1).

Figura 8 - Comparação entre Pi 3 e Zero W

Fonte: Null Byte (2017, p. 1)

Conforme a Figura 8, o Raspberry Pi Zero e Zero W são a metade do tamanho de um modelo Pi 3, com um único núcleo de 1 GHz, 512 MB de RAM, porta mini High-Definition Multimedia Interface (HDMI) e USB. Além disso ele integrou *wireless* 802.11n e Bluetooth na placa com a versão Pi Zero W (RASPBERRY PI FONDATION, 2017, p. 1).

2.4 TRABALHOS CORRELATOS

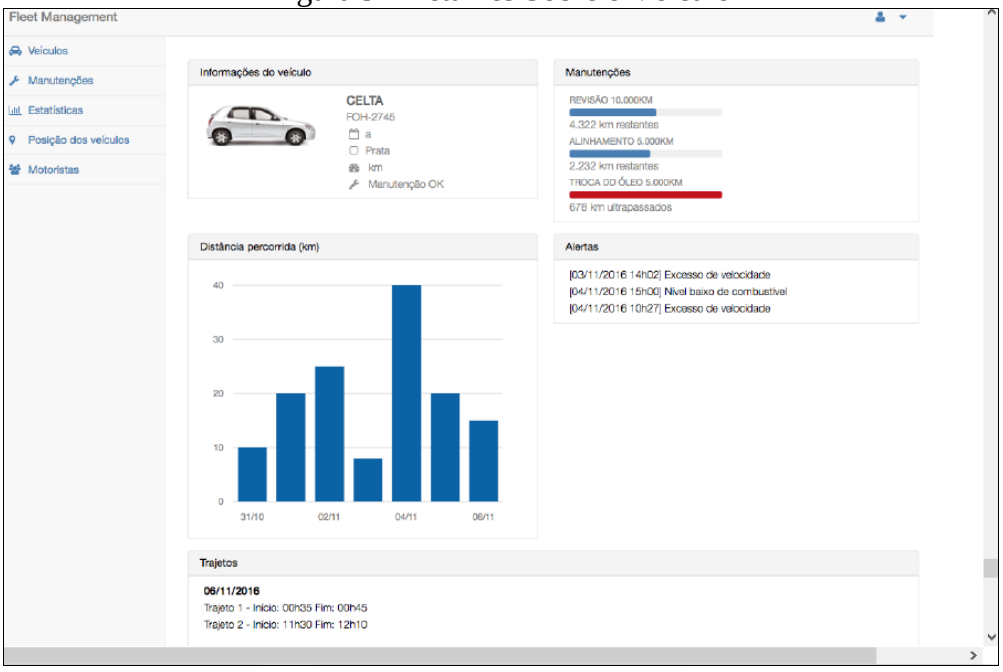
São apresentados dois trabalhos correlatos que possuem características semelhantes à este desenvolvido. Primeiramente, seção 2.4.1 apresenta uma solução para gestão de frotas denominado Gestão de uma Frota de Veículos Utilizando Sistemas Embarcados desenvolvido por Pacheco (2016). Por fim, a seção 2.4.2 trata de uma aplicação chamada Sistema de Localização de Veículos Para Smartphone Android feita por Pina (2015).

2.4.1 Gestão de frota de veículos

Segundo Pacheco (2016), o objetivo do trabalho é o desenvolvimento de um sistema embarcado para o monitoramento de veículos integrado a uma plataforma *web* para o gerenciamento de frota de veículos. Foi desenvolvido um sistema embarcado construído para uma placa Raspberry Pi, um adaptador OBD e um módulo GPS. A placa Raspberry Pi é um computador que utiliza o sistema operacional Linux que ocupa o papel central do sistema. Através dela é possível coletar os dados do adaptador OBD e do módulo GPS. Além disso, ela transmite os dados ao servidor que hospeda a uma plataforma *web*.

Pacheco (2016) desenvolveu um sistema embarcado tal que, dentro do contexto de gestão de frotas, pudesse coletar informações através da porta OBD, como por exemplo: velocidade do veículo, rotação do motor, carga do motor, distância percorrida e geolocalização. Essas informações foram coletadas para analisar como o veículo é conduzido (conforme Figura 9). O monitoramento pode indicar se os limites de velocidade são respeitados e se há acelerações e frenagens bruscas. Valores de rotação muito elevados também seriam detectados. A carga do motor está associada ao consumo de combustível. A distância percorrida para o agendamento sem a necessidade de consultar o odômetro, a Figura 9 mostra na parte de manutenções essas informações prévias de quilometragem para manutenção, troca de óleo e alinhamento do veículo. Por fim, utiliza a geolocalização para registrar os trajetos realizados.

Figura 9 - Detalhes Sobre o Veículo



Fonte: Pacheco (2016, p. 70)

Estes dados são coletados e armazenados no cartão de memória da placa Raspberry Pi. Para isso, foi considerado um intervalo de amostragem desta coleta. Após a coleta dos dados, eles são enviados à um servidor que hospeda a plataforma de gestão de frotas. O envio de dados é feito através de uma conexão Wi-Fi no momento que o veículo retorna à garagem.

Pacheco (2016) utilizou a linguagem de programação Python orientado à objetos. Ele cita que Python é uma linguagem de programação que tem uma quantidade considerável de bibliotecas disponíveis e podem ser desenvolvidas aplicações, tais como: aplicações *web*, cálculo científico e numérico, gráficos, *Machine Learning*, *Data Science*, visualização de dados, interfaces gráficas, entre outras. Foram utilizadas neste trabalho bibliotecas que

permitissem a criação de *threads*, o acesso ao banco de dados SQLite, o uso de expressões regulares e o *back-end* de um servidor. Para o banco de dados, foi utilizado o SQLite por ser possível executar comandos SQL, salvar e fazer alterações em registros.

Para comandar o sistema embarcado, foi escolhida a placa Raspberry Pi 3 Model B, a qual funciona com o sistema operacional Linux. Possui Bluetooth e uma porta serial, podendo assim, comunicar-se com o adaptador OBD e com o módulo GPS.

O adaptador OBD é o dispositivo utilizado para a comunicação com o computador de bordo do carro e é instalado diretamente na porta OBD do veículo. A troca de dados entre a Raspberry Pi é realizada através de Bluetooth. O adaptador utilizado é baseado no circuito integrado ELM327. O ELM327 é um interpretador multiprotocolo projetado para funcionar com todos os protocolos automotivos de veículos previstos na especificação OBD2. Este dispositivo custa aproximadamente trinta reais. Foi utilizado também o módulo GPS GY-NEO6VM2 para determinar a posição dos veículos utilizando a porta serial para fazer a comunicação dele com a placa Raspberry Pi.

Pacheco (2016) cita que a solução mostrou-se eficiente na coleta de dados, pois irá trabalhar com informações atualizadas lidas direto dos sensores do veículo, bem como elimina a necessidade da leitura constante do odômetro do veículo. O esforço das empresas que possuem um número elevado de veículos, diminui pelo fato do sistema executar a leitura dos sensores e também melhora a capacidade de detecção para manutenção da frota (PACHECO, 2016).

Pacheco (2016) sugere que sejam implementados soluções para veículos pesados, pois a solução desenvolvida por ele abrange somente veículos de passeio, em razão de que a interface do sistema OBD é diferente da implementada por ele. O autor destaca a necessidade da portabilidade da plataforma *web* para um sistema *mobile*. Sugere-se também, uma redução de custo da placa, utilizando ao invés da placa Raspberry Pi 3 (que custa 35 dólares aproximadamente), uma placa Raspberry Pi Zero (que custa 5 dólares) que atende a especificação e ainda consome menos energia.

2.4.2 Localização de veículos para Android

Pina (2015) cita que o trabalho consiste no desenvolvimento de um sistema de localização de veículos para *smartphone* Android. Para isso, foram desenvolvidas duas aplicações: uma aplicação de localização Android e uma aplicação *web* para monitoramento.

A aplicação de localização permite capturar dados de localização de GPS e estabelecer uma rede *piconet* Bluetooth, admitindo assim uma comunicação com uma unidade de controle

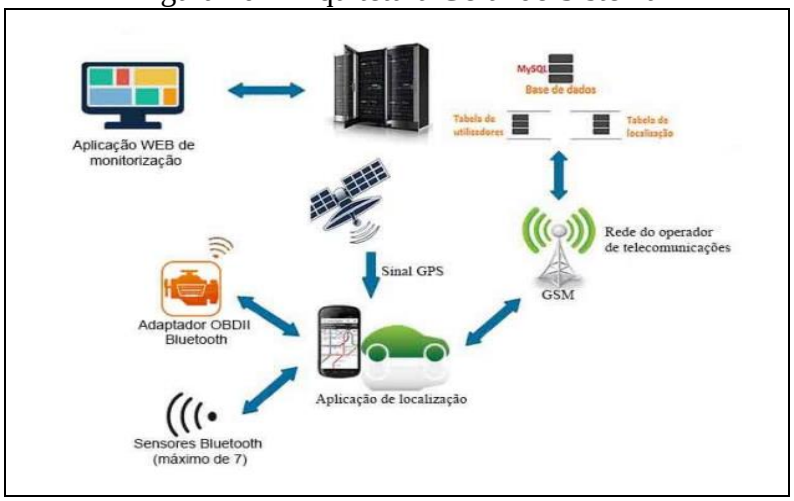
do carro através de um adaptador OBD2 e com até sete sensores/dispositivos Bluetooth que podem ser instalados no veículo. Os dados adquiridos pela aplicação Android são enviados periodicamente para um servidor *web*.

A aplicação *web* desenvolvida por Pina (2015), permite ao gestor da frota, efetuar o monitoramento dos veículos em circulação registrados no sistema. É possível visualizar a posição geográfica dos veículos em um mapa, bem como, os dados do mesmo e sensores/dispositivos Bluetooth para cada localização enviada pela aplicação Android.

A Figura 10 exemplifica a arquitetura geral do sistema desenvolvido por Pina (2015), ele idealiza o sistema em quatro principais componentes:

a) aplicação Android de localização;
b) aplicação *web* de monitoramento;
c) adaptadores OBD2/Bluetooth;
d) dispositivos/sensores Bluetooth.

Figura 10 - Arquitetura Geral do Sistema

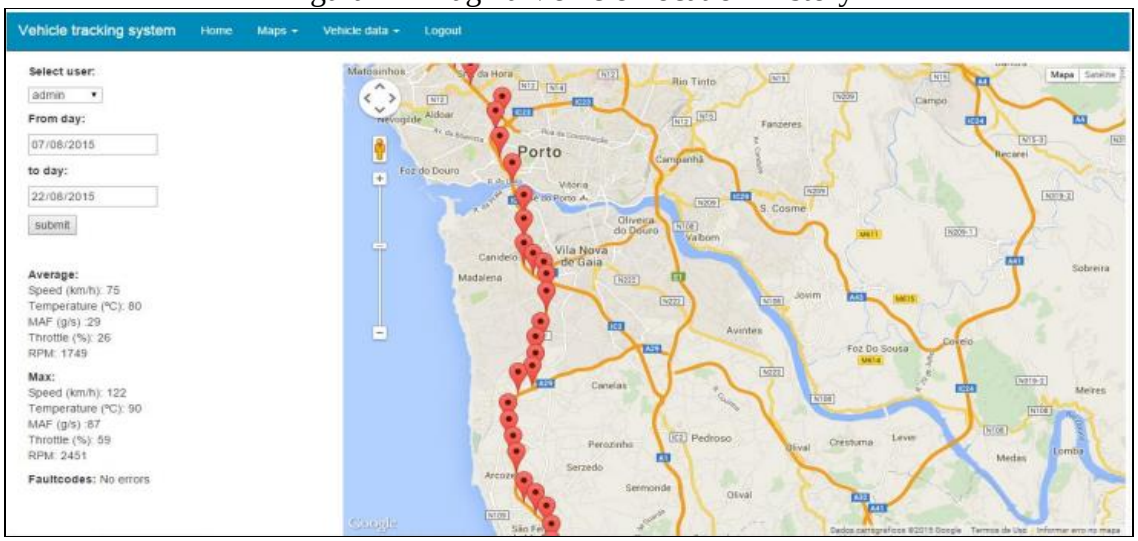


Fonte: Pina (2015, p. 79)

A aplicação Android desenvolvida é responsável por gerir a comunicação com o adaptador OBD2/Bluetooth, possíveis sensores Bluetooth adicionais, posição geográfica e transmitir esses dados para o servidor *web*. Para essa aplicação, utilizou-se JAVA com Android e para a persistência de dados no aparelho, utilizou-se o banco de dados SQLite.

Já a aplicação *web* foi desenvolvida para verificar onde está cada veículo geograficamente e as informações capturadas das portas OBD2 e sensores Bluetooth. Na Figura 11, pode-se observar a funcionalidade de visualizar as informações geográficas em um mapa. Essa aplicação *web* foi desenvolvida utilizando a tecnologia Personal Home Pages (PHP) juntamente com o banco de dados MySQL para persistência.

Figura 11 - Página Vehicle Location History



Fonte: Pina (2015, p. 115)

Pina (2015) conclui que o sistema desenvolvido é completamente funcional e teve alguma complexidade com o desenvolvimento na plataforma Android, mas que possui uma documentação completa. O autor cita também, que teve facilidade em integrar os dispositivos devido à simplicidade de instalação e portabilidade. A possibilidade de interação com o adaptador OBD2 Bluetooth e com até sete dispositivos simultaneamente torna o sistema extremamente versátil, que pode ser utilizada à inúmeras aplicações, até mesmo fora do contexto de veículos como foi apresentado.

Pina (2015) relata que uma das desvantagens foi a fragilidade do adaptador OBD2/Bluetooth, sugere então, que seja utilizado outro com maior qualidade. O sistema pode aumentar a quantidade de funcionalidades disponíveis, tais como: a leitura de mais parâmetros da porta OBD2, melhoras no layout da aplicação e também um sistema de mensagens entre gestor e utilizador da aplicação.

2.5 FERRAMENTAS ATUAIS

Nesta seção serão apresentados dois trabalhos, ambos desenvolvidos pelo curso de Ciência da Computação na Universidade Regional de Blumenau. A seção 2.5.1 trata de um trabalho denominado Findcar desenvolvido por Baumgarten (2016). Por fim, a seção 2.5.2 apresenta o trabalho OBD-JRP, desenvolvido por Starosky (2016).

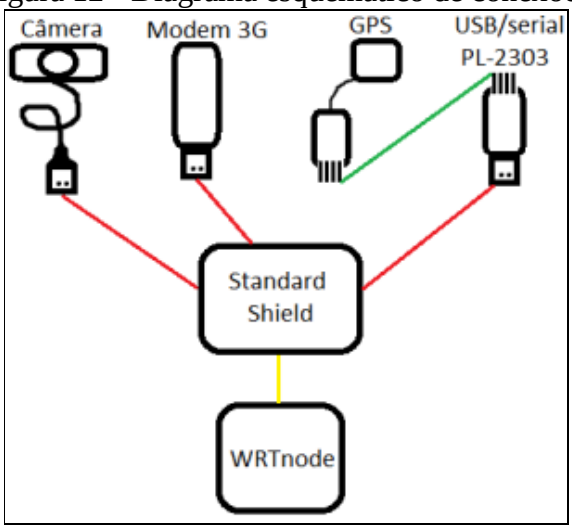
2.5.1 FINDCAR

Baumgarten (2016) desenvolveu um dispositivo que possibilitasse o rastreamento veicular através de geolocalização e uma imagem capturada dor meio de uma câmera acoplada neste dispositivo. Os objetivos cumpridos no trabalho foram:

a) realizou-se a integração do OpenWRT com: um modem 3rd Generation (3G), um módulo Global Positioning System (GPS) e uma câmera;
b) desenvolveu uma plataforma *web* para verificar a localização atual, ultimas localizações do veículo, capturar imagens e configurar o envio de notificações por e-mail;
c) tornou o rastreador o próprio servidor onde a aplicação é executada.

Para este rastreador veicular, Baumgarten (2016) utilizou uma placa WRTnode de modelo MT7620 com OpenWRT, que é uma distribuição customizável do Linux para sistemas embarcados. Utilizou também, um módulo de GPS Ublox GY-NEO6MV2 para capturar a geolocalização e uma webcam Logitech C270 para obter as imagens do veículo. Todas as informações são disponibilizadas através de um modem 3G/4G Huawei E3272. O esquema de conexões pode ser observado na Figura 12.

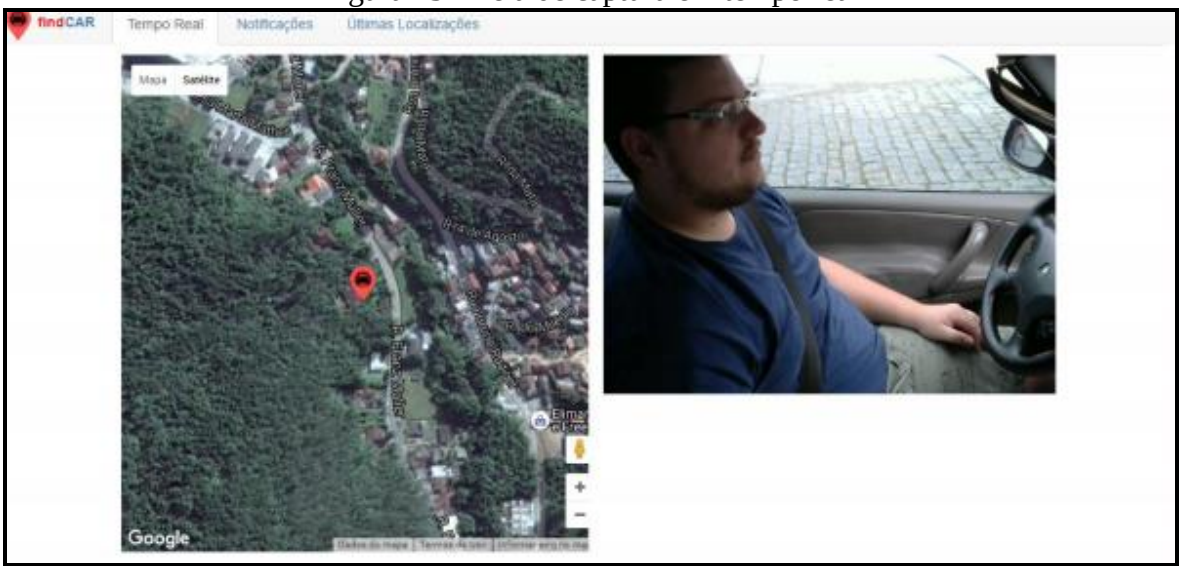
Figura 12 - Diagrama esquemático de conexões



Fonte: Baumgarten (2016, p. 26)

Para desenvolver a aplicação *web*, Baumgarten (2016) utilizou para interface gráfica HTML5, CSS3 e Bootstrap3. Com relação à comunicação com o servidor, foi utilizado PHP e a persistência de dados foi feita através do banco de dados MySQL. Utilizou também Google Maps Javascript Api para mostrar as coordenadas capturadas pelo GPS no mapa, conforme Figura 13.

Figura 13 - Tela de captura em tempo real



Fonte: Baumgarten (2016, p. 39)

Baumgarten (2016) ressalta que o objetivo do trabalho foi atingido adequadamente. O uso da linguagem PHP supriu as necessidades do sistema. Ele enfatiza o uso do banco de dados MySQL que foi facilmente integrado e manipulado com o OpenWRT.

2.5.2 OBD-JRP

Staroski (2016) desenvolveu um protótipo de software embarcado em uma placa Raspberry Pi. Esta placa foi utilizada para capturar dados da porta OBD de um veículo e disponibilizá-los em uma página *web*. Ele enumerou e concluiu alguns objetivos específicos que foram atendidos, são eles:

a) desenvolver o firmware, que irá monitorar a porta OBD2 do carro, coletar dados e os enviar para um servidor;
b) desenvolver o software servidor, que irá receber os dados coletados pelo firmware e armazenar os mesmos;
c) desenvolver uma página *web* para consultar o histórico dos dados.

Para a elaboração do trabalho, Starosky (2016) utilizou o ambiente de desenvolvimento Java com a biblioteca BlueCove para realizar a comunicação com a interface ELM327 Bluetooth. No desenvolvimento do servidor, foi utilizado a biblioteca Google Charts para criar gráficos com a linguagem Javascript. Foi utilizado a placa Raspberry Pi 3 Model B com o sistema operacional Raspian GNU/Linux 8 que é disponibilizada com a versão 1.8 do Java. Os dispositivos citados e utilizados podem ser visualizados na Figura 14. Além desses dispositivos, para concluir a comunicação com o servidor, foi utilizado um modem 3G/4rd Generation (4G) da marca Huawei.

Figura 14 - Instalação no Volkswagen SpaceFox 2009

Fonte: Starosky (2016, p. 73)

Segundo o autor, a aplicação foi testada em três veículos, sendo eles: GM Corsa Sedan 2003, Volkswagen Gol 2010 e um Volkswagen SpaceFox 2009. Todos possuíam o conector OBD2, entretanto o Corsa Sedan 2003 não implementava nenhum protocolo OBD2 apesar de possuir a porta. O protótipo atendeu os objetivos propostos e o Raspberry Pi atendeu as exigências computacionais desenvolvidas.

3 DESENVOLVIMENTO DA APLICAÇÃO

--MAKE

3.1 ESPECIFICAÇÃO

Nesta seção é exposta a especificação da aplicação através de requisitos e diagramas. Inicialmente são apresentados os Requisitos Funcionais (RF), Requisitos Não Funcionais (RNF) e sua rastreabilidade com cada caso de uso. Em seguida é exposto o diagrama de casos de uso, também é apresentado diagramas sobre a arquitetura de hardware, e por fim diagramas de atividades. Todos os diagramas Unified Modeling Language (UML) foram construídos utilizando a ferramenta online Draw.io. O diagrama para esquema de componentes eletrônicos utilizou a ferramenta online denominada Scheme-It para o seu desenvolvimento.

3.1.1 Requisitos

O Quadro 7 apresenta os principais requisitos funcionais (RF) da aplicação e sua respectiva rastreabilidade com cada caso de uso correspondente.

Quadro 7 - Requisitos funcionais da aplicação e rastreabilidade

Requisitos funcionais	Caso de uso
RF01: permitir que o servidor embarcado seja iniciado automaticamente ao ligar a placa Raspberry Pi W Zero	UC01
RF02: permitir ao usuário fazer a configuração de e-mail, telefone e notificações	UC02
RF03: permitir que o usuário capture imagens em tempo real do interior do veículo	UC03
RF04: disponibilizar informações da geolocalização do veículo	UC04
RF05: disponibilizar dados de sensores do automóvel através da porta OBD2	UC05
RF06: disponibilizar os códigos de erro (DTCs)	UC06
RF07: permitir limpar os códigos de erro (DTCs)	UC06
RF08: permitir consultar o status da luz de mal funcionamento (MIL)	UC06
RF09: permitir receber notificação via e-mail e SMS caso ocorram falhas (DTCs) no veículo	UC08
RF10: permitir reiniciar o servidor embarcado através de um botão acoplado a placa Raspberry PI Zero W	UC07
RF11: permitir desligar e ligar o sistema operacional embarcado através de um botão acoplado a placa Raspberry PI Zero W	UC01
RF12: permitir visualizar estágios de execução do servidor embarcado através de Diodos Emissores de Luz (Light Emitting Diode – LED)	UC09

Fonte: elaborado pelo autor.

Após a exposição dos requisitos funcionais, são apresentados abaixo os principais requisitos não funcionais (RNF) previstos para a aplicação desenvolvida. A aplicação deve:

a) utilizar a placa Raspberry PI Zero W;
b) permitir integrar a placa Raspberry PI Zero W com o conector padrão SAE J1962 do veículo via Bluetooth através de um adaptador ELM327;

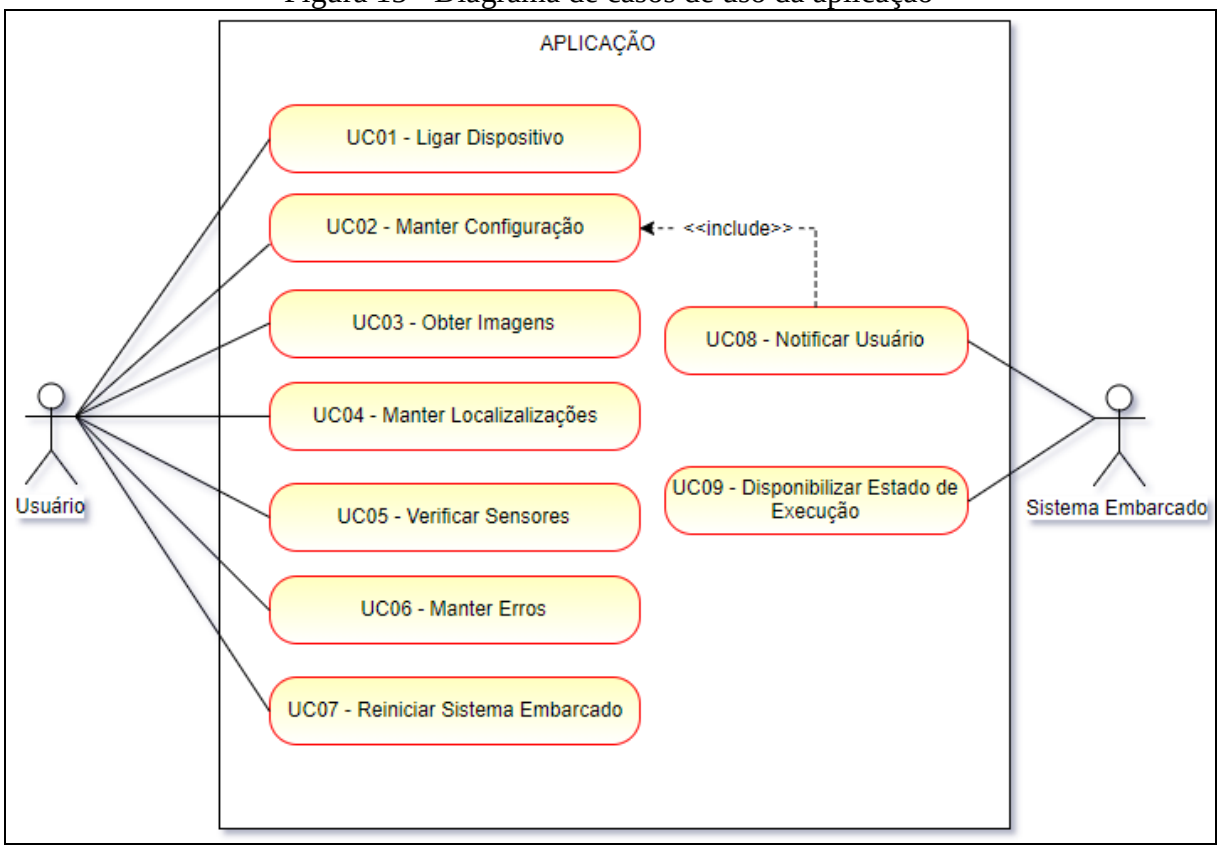
c) permitir integrar a placa Raspberry PI Zero W com o módulo GPS Ubox GY-GPS6MV2;
d) permitir integrar a placa Raspberry PI Zero W com o módulo WAVGAT Raspberry Pi Camera Revision 1.3;
e) utilizar a operadora TIM com o modem USB 3G da marca ZTE MF626;
f) utilizar o sistema operacional Raspian instalado na placa Raspberry Pi Zero W;
g) permitir que o sistema embarcado persista dados em formato JavaScript Object Notation (JSON), sem a necessidade de banco de dados;
h) permitir que a aplicação persista dados em formato JSON na memória do aplicativo *mobile* através do banco de dados IndexedDB.
i) utilizar a linguagem de programação Python para o sistema embarcado;
j) utilizar a linguagem de programação Typescript para o aplicativo *mobile*;
k) utilizar a biblioteca Flask como servidor de aplicações *web*;
l) utilizar software OBDSim e ECU Engine Pro para a simulação de uma central automotiva;
m) utilizar a biblioteca Ionic Framework para desenvolver a aplicação *mobile*;
n) utilizar a ferramenta Draw.io para a modelagem de diagramas UML;
o) utilizar a ferramenta Scheme-it para desenvolver o diagrama de esquema eletrônico.

3.1.2 Diagrama de casos de uso

Nesta sessão é apresentado o diagrama de casos de uso (exibido na Figura 15). No total, foram necessários nove casos de uso que representam as principais funcionalidades da aplicação desenvolvida. Conforme pode-se observar, foram identificados dois atores, são eles:

a) `Usuário`: ator responsável por comandar a aplicação;
b) `Sistema Embarcado`: este ator é designado à dar *feedback* do dispositivo e notificar alguma falha (caso existir) no OBD2.

Figura 15 - Diagrama de casos de uso da aplicação



Fonte: elaborado pelo autor.

No caso de uso `UC01 - Ligar Dispositivo`, é possível que o usuário ligue o dispositivo e desligue-o através de um botão. O caso de uso `UC02 - Manter Configuração` destina-se para que o usuário possa fazer a configuração de e-mail, telefone e se deseja receber as notificações emitidas caso houverem falhas DTC. Já o `UC03 - Obter imagens` propõe-se a capturar imagens do dispositivo, elas podem ser salvas na aplicação *mobile* ou visualizadas através de *stream* em tempo real. No caso de uso `UC04 - Manter Localizações`, o usuário poderá capturar a geolocalização do veículo e também é permitida a visualização das últimas localizações solicitadas. No caso de uso `UC05 - Verificar Sensores` o usuário pode visualizar dados dos sensores do automóvel em tempo real. O caso de uso `UC06 - Manter Erros` possibilita que o usuário visualize dados de códigos de erro (DTC) e estado da lâmpada MIL, além disso, este caso de uso permite a limpeza dos DTCs. Por fim, no `UC07 - Reiniciar Sistema Embarcado` o usuário pode fazer a reinicialização do sistema embarcado no dispositivo.

Já o `Sistema Embarcado` tem acesso à dois casos de uso, são eles:

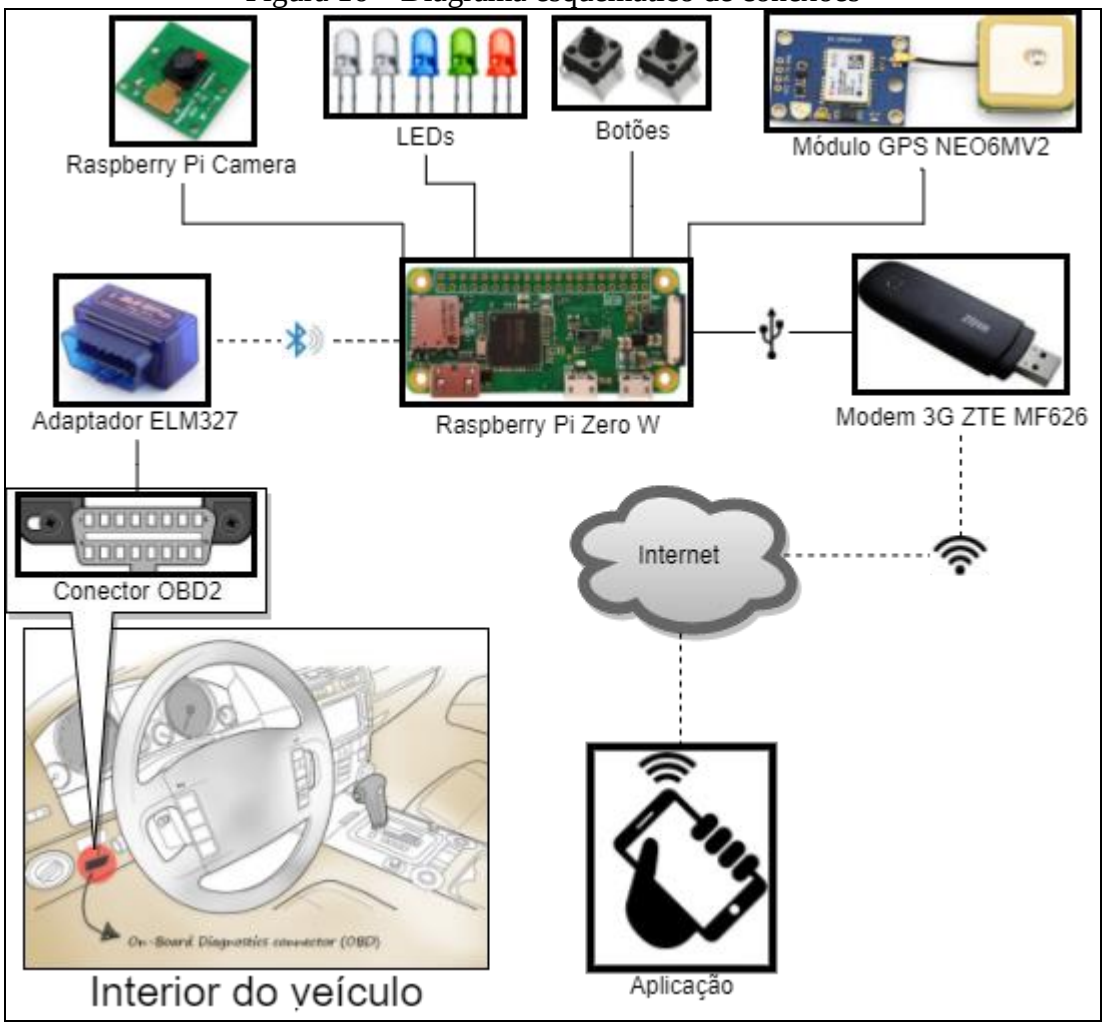
a) `UC08 - Notificar Usuário`: permite que o sistema embarcado notifique o usuário se ocorrer algum erro DTC, essa notificação é configurada previamente pelo `Usuário` no `UC02`;

b) `UC09 – Disponibilizar Estado de Execução`: permite que o sistema embarcado possa fornecer o estado de execução atual para o `Usuário` através de LEDs.

3.1.3 Diagrama de arquitetura da aplicação

Nesta seção, é exposta a arquitetura de hardware aplicada no desenvolvimento deste trabalho. Foi elaborado um diagrama para apresentar as ligações de componentes eletrônicos utilizados. A Figura 16 ilustra o diagrama esquemático de conexões realizadas e seus respectivos meios de comunicação.

Figura 16 – Diagrama esquemático de conexões



Fonte: elaborado pelo autor.

Pode-se observar que a placa `Raspberry Pi Zero W` é o *core* do projeto. Todas as principais conexões passam por ela até a disponibilização das informações na aplicação *mobile*.

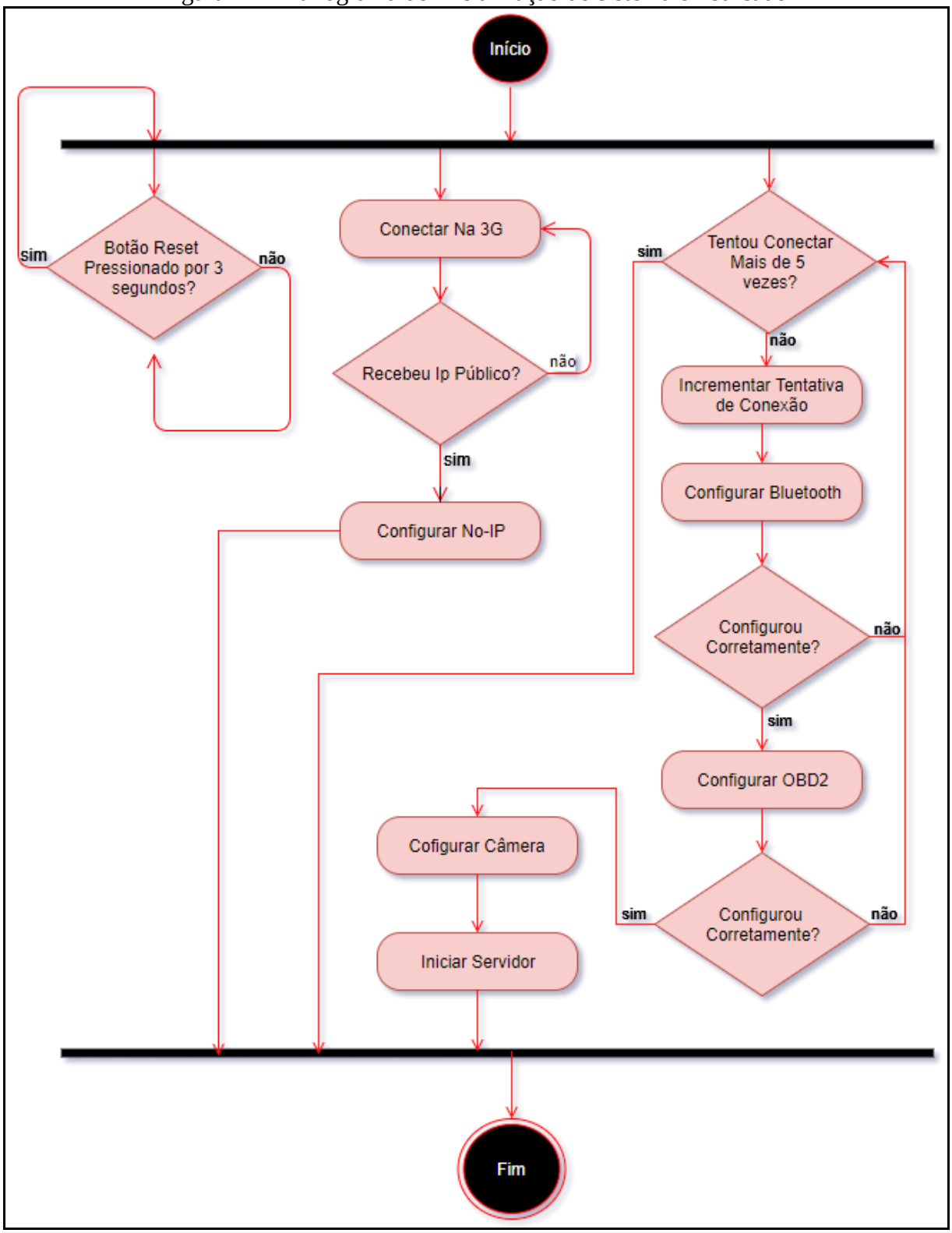
3.1.4 Diagramas de atividades

Na Figura 17 é representado o processo de inicialização do sistema embarcado na placa Raspberry Pi Zero W através de um diagrama de atividades. Este engloba o fluxo que é executado após a ligação da placa na energia, inicialização ou reinicialização do sistema operacional.

Na ativação do sistema operacional, executa-se um processo que inicializa o sistema embarcado, este processo obedece os seguintes passos paralelamente:

a) espera que o usuário pressione o botão de reinicialização durante 3 segundos, caso isso aconteça, o processo é reiniciado;
b) executa a discagem para conectar-se com a rede 3G até que receba um IP público, se obtiver sucesso, configura-se o serviço de Dynamic Domain Name System (DDNS) No-Ip com o novo endereço público;
c) tenta durante cinco vezes configurar as conexões Bluetooth e OBD2, caso consiga configurar o Bluetooth, o próximo passo a ser executado é a configuração OBD2. Se a conexão Bluetooth e OBD2 forem configuradas com sucesso, inicializa-se a preparação do módulo da câmera. Por fim, o servidor embarcado é ativado.

Figura 17 - Fluxograma de inicialização do sistema embarcado

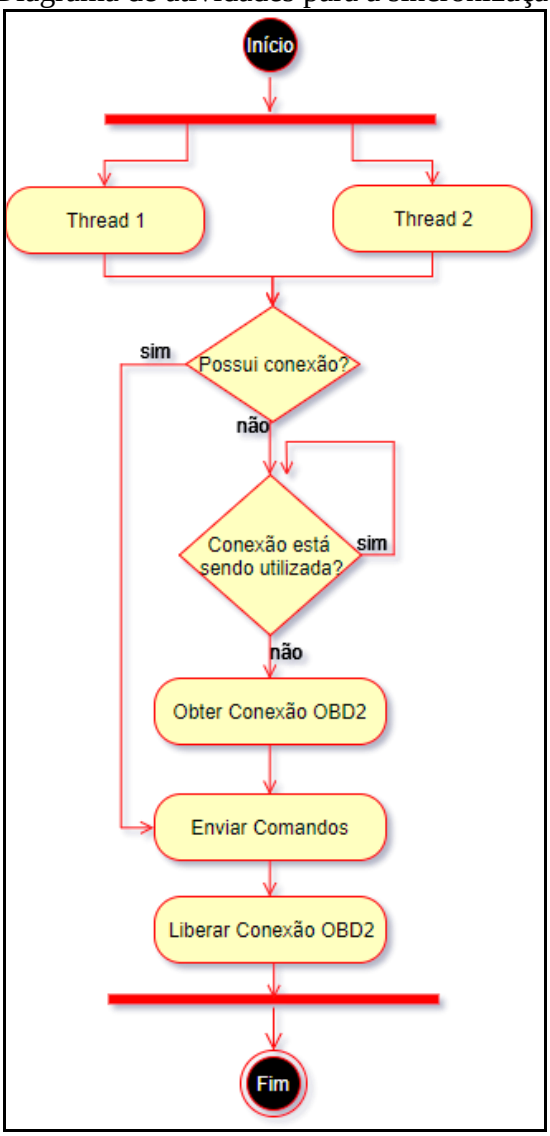


Fonte: elaborado pelo autor.

Na Figura 18 é ilustrado o diagrama de atividades relacionado ao fluxo de sincronização de threads utilizado pelo sistema embarcado. O sistema embarcado inicia duas *threads* que são: `Thread 1` – responsável pela leitura dos sensores OBD2, `Thread 2` –

monitora os dados de códigos de erros DTC a cada 20 segundos. Ambas as threads só podem obter a conexão com a porta OBD2 caso ela não esteja sendo utilizada.

Figura 18 - Diagrama de atividades para a sincronização de *threads*



Fonte: elaborado pelo autor.

3.2 IMPLEMENTAÇÃO

--MAKE

A seção 3.2.1 tem como objetivo a apresentação e os detalhes da construção do dispositivo de hardware desenvolvido para o trabalho, bem como, os componentes utilizados e suas respectivas funções destinadas.

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

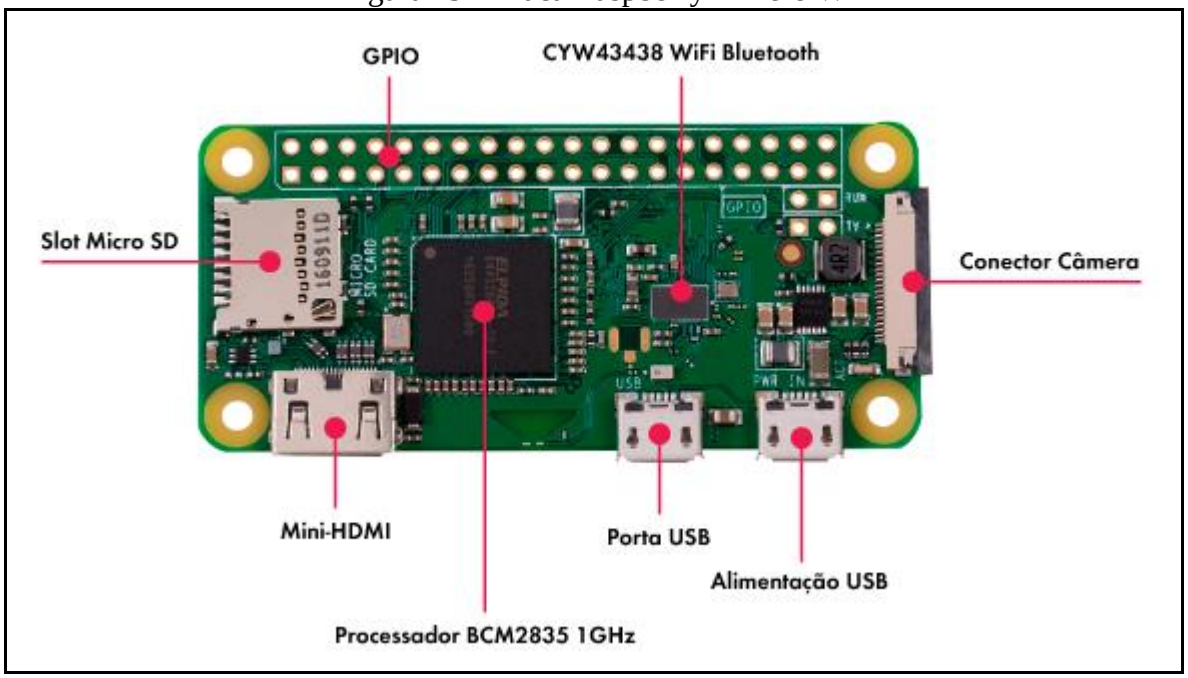
3.2.1 Hardware e periféricos da aplicação

Para a construção do dispositivo de hardware foi necessária a placa Raspberry Pi Zero W, um módulo WAVGAT Raspberry Pi Camera versão 1.3, um cabo flexível e um módulo GPS Ubox GY-GPS6MV2. Também foi utilizado uma placa universal, dois botões *switch*, fios, barramento para *jumpers*, cinco resistores 220 ohms e cinco LEDs. Além do hardware desenvolvido, foram necessários: um adaptador ELM327, um modem USB ZTE MF626, um chip de telefonia móvel TIM e por fim, um cartão micro SanDisk (SD) de 16 GB. A relação de custo para todas as peças e componentes utilizados no projeto estão descriminados no Apêndice A.

Segundo Thomsen (2017), a placa Raspberry Pi Zero W (ilustrada na a Figura 20) foi lançada no dia 28 de fevereiro de 2017 como sendo uma atualização da versão Raspberry Pi Zero. Thomsen (2017, p. 1) ainda ressalta que: “A Raspberry Pi Zero W usa o mesmo chip Cypress CYW43438 sem fio que a Raspberry Pi 3 e incorpora um novo layout de antena PCB licenciado pela empresa sueca ProAnt. A potência da placa continua a mesma [...]”. A placa foi selecionada para o desenvolvimento da aplicação principalmente por disponibilizar:

a) chip CYW43438 *wireless* Bluetooth 4.1 integrado para a comunicação com o adaptador OBD2;
b) porta Camera Serial Interface (CSI) para ligar o módulo WAVGAT Raspberry Pi Camera versão 1.3;
c) porta USB para conectar o modem USB ZTE MF626.
d) barramento General Purpose Input/Output (GPIO) para a ligação dos demais componentes que são: módulo GPS Ubox GY-GPS6MV2, LEDs e botões.

Figura 19 – Placa Raspberry Pi Zero W

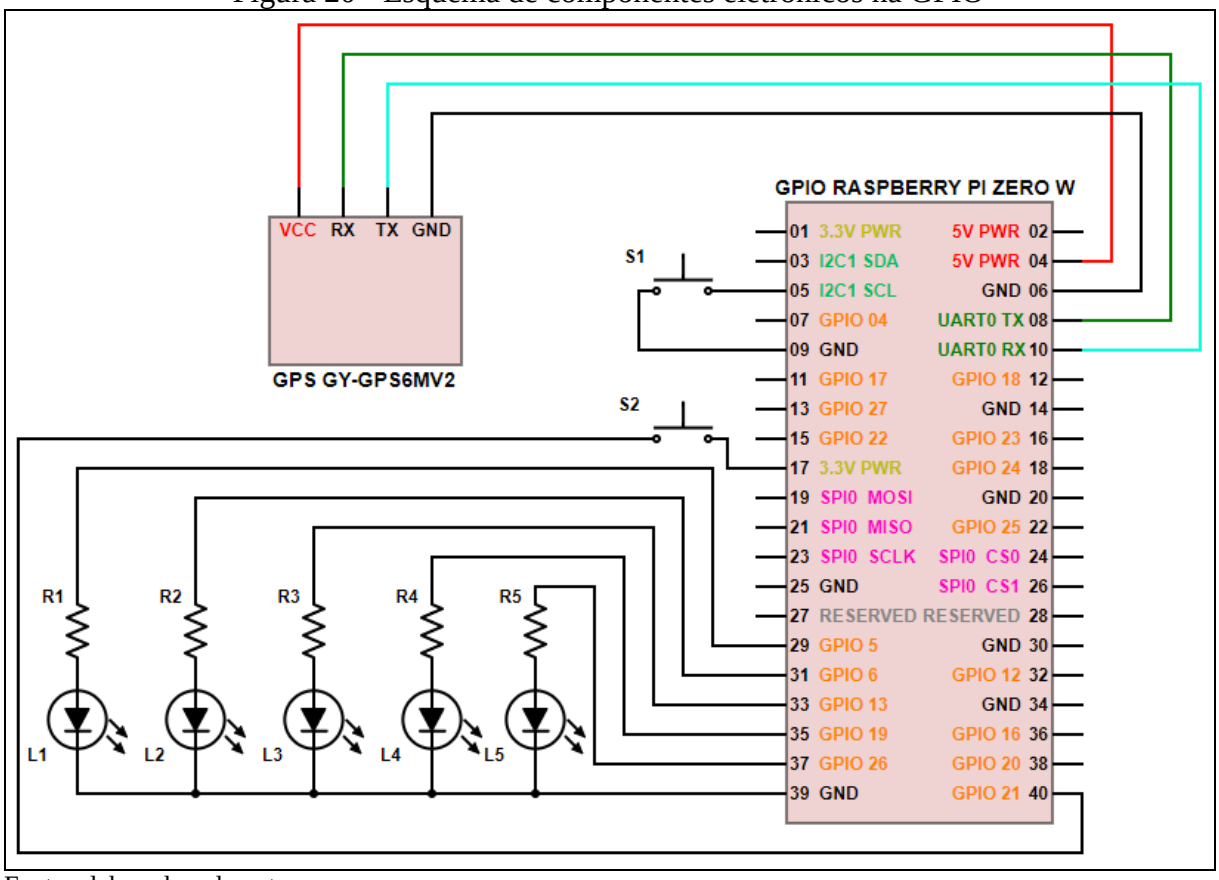


Fonte: Thomsen (2017, p. 1).

Além dos principais recursos utilizados pelo trabalho, a placa Raspberry Pi Zero W conta com um processador Broadcom BCM2385 1 GHz de único núcleo CPU, 512 MB de RAM e uma porta micro SD. Sua alimentação é feita através de uma porta USB exclusiva para este fim, porém, isso pode ser feito pelas duas portas.

A Figura 20, trata-se da apresentação do esquema de componentes eletrônicos conectados através do barramento GPIO da placa Raspberry Pi Zero W. Os principais componentes ligados ao barramento são os LEDs, os botões e o módulo GPS Ubox GY-GPS6MV2. Este esquema de componentes foi desenvolvido através da ferramenta online denominada Scheme-it.

Figura 20 - Esquema de componentes eletrônicos na GPIO



Fonte: elaborado pelo autor

Na Figura 20, o botão S1 é ligado ao pino 09 (Ground - GND) e ao pino 05 (IC21 SCL). Já botão S2 é conectado ao pino 40 do barramento (GPIO 21) e ao pino 17 (3.3V). Os LEDs estão ligados à resistores de 220 ohms e conectados ao pino 39 (GND). A placa contém os seguintes resistores e suas respectivas conexões:

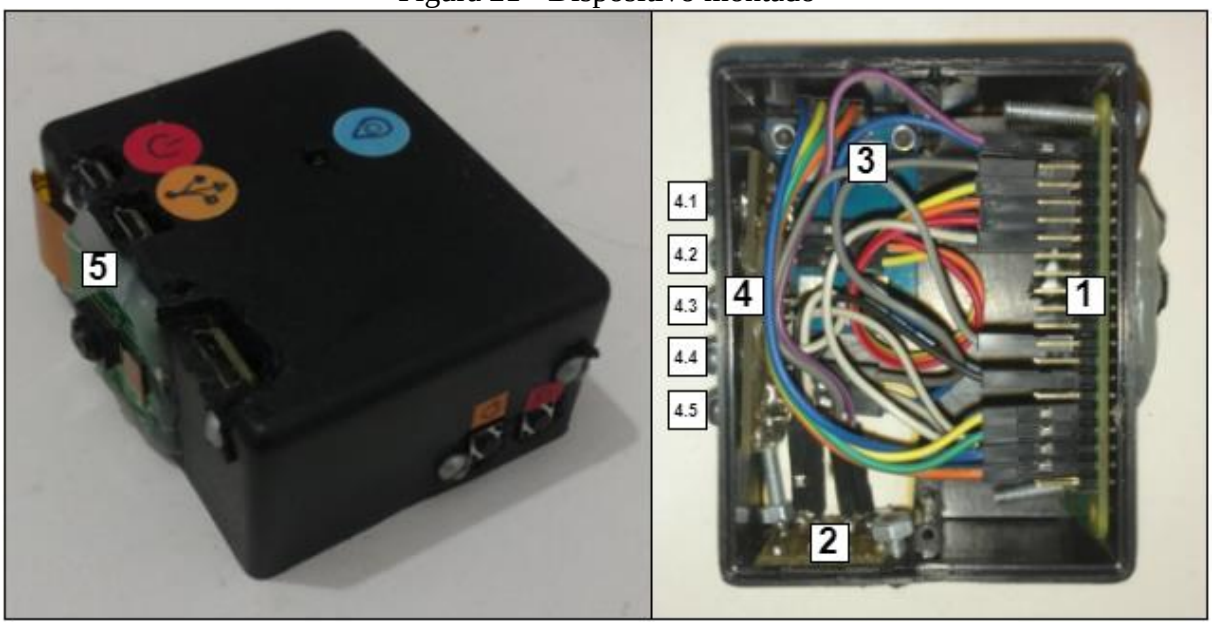
a) R1: L1 e pino 29 (GPIO 5);
b) R2: L2 e pino 31 (GPIO 6);
c) R3: L3 e pino 33 (GPIO 13);
d) R4: L4 e pino 35 (GPIO 19);
e) R5: L5 e pino 37 (GPIO 26).

Por fim, esquema da Figura 20 apresenta a conexão do módulo GPS Ubox GY-GPS6MV2 na GPIO da placa Raspberry Pi Zero W. O RX do módulo é ligado ao pino UART0 TX (pino 08) da placa e o TX do módulo GPS é ligado ao UART0 RX (pino 10) da placa, o VCC é ligado ao pino 04 (5.5V PWR) e o GND ao pino 06.

Na Figura 21 é exposto o dispositivo elaborado para o trabalho. Foi utilizado uma caixa para plástica para proteger os componentes de hardware e facilitar o seu transporte. Este

dispositivo possui 7 cm de largura, 5 cm de comprimento e 3 cm altura. A caixa foi escolhida por ser de menor tamanho encontrado no site da Proesi e que acoplasse todos os componentes.

Figura 21 - Dispositivo montado



Fonte: elaborado pelo autor

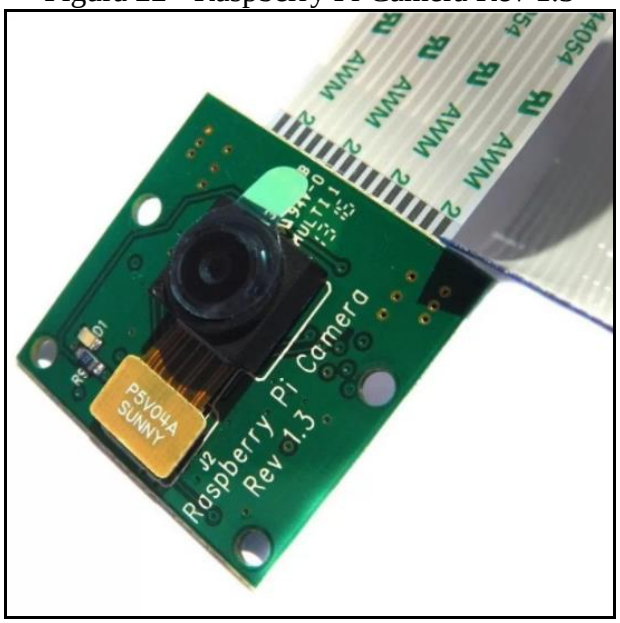
O dispositivo organiza todos os componentes que são conectados através do barramento GPIO da placa Raspberry Pi Zero W. Também é ligado à ele o modulo WAVGAT Raspberry Pi Camera. A Figura 21 dispõe de:

a) item 1: Placa Raspberry Pi Zero;
b) item 2: botões;
c) item 3: módulo GPS Ubox GY-GPS6MV2;
d) item 4: LEDs sendo de cor:
   - item 4.1 e 4.2: branca,
   - item 4.3: azul,
   - item 4.4 verde,
   - item 4.5 vermelha.
e) item 5: módulo WAVGAT Raspberry Pi Camera.

A câmera utilizada para o trabalho foi a WAVGAT Raspberry Pi Camera Rev 1.3 (ilustrada Figura 22) conectada na entrada CSI da placa Raspberry Pi Zero W através de um cabo flexível. Segundo Pi Supply (2017), a câmera possui uma resolução de 5 Megapixel e

grava vídeos em Hight Definition (HD) de 1080p[1] à 30 quadros por segundo (fps). Ainda, o autor afirma que este módulo oferece suporte aos modelos Rasberry Pi A e B.

Figura 22 - Raspberry Pì Camera Rev 1.3



Fonte: Pi Supply (2017, p. 1).

A Figura 23 ilustra o adaptador ELM327. A comunicação entre o adaptador ELM327 e a placa Raspberry Pi Zero W é feita através de Bluetooth.

Figura 23 - Adaptador ELM327 Bluetooth

Fonte: AliExpress (2017, p. 1).

Na Figura 24, é apresentado o modem USB. Segundo ZTE Brasil (2012), este dispositivo da marca ZTE de modelo MF626 trabalha a com frequência de 2rd Generation

[1] É o nome abreviado de um tipo de resolução de imagem de telas ou monitores. O número 1080 representa 1080 linhas horizontais de resolução vertical, enquanto a letra p denota uma varredura progressiva (GIZMODO, 2013).

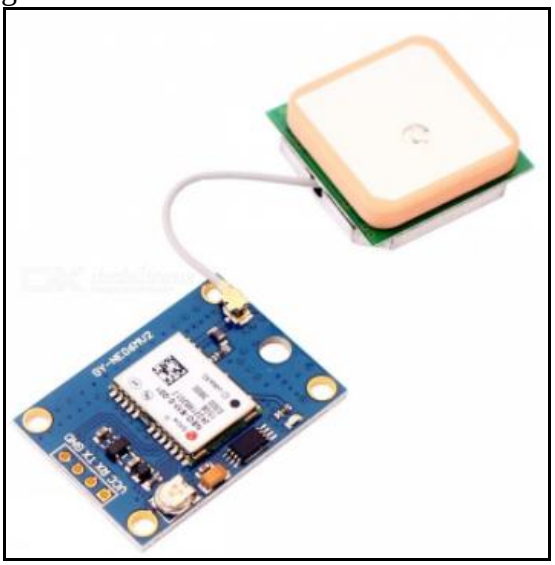
(2G) e de 3rd Generation (3G) e sua taxa de transferência é de 3.6 Megabits por segundo (Mbps) quando conectado na rede de telefonia móvel 3G. O autor afirma que com este dispositivo é possível mandar mensagens Short Message Service (SMS) e o modem também possui uma porta para a leitura de cartões SD de até 4 GB. Este modem conecta-se com a placa Raspberry Pi Zero W através de USB com um conversor de micro USB e utiliza um chip da operadora TIM pré-pago para conectar-se à Internet.

Figura 24 - Modem 3G ZTE MF626

Fonte ZTE Brasil (2012).

Por fim, na Figura 25 é ilustrado o módulo GPS Ubox GY-GPS6MV2. Ele é conectado através de GPIO na placa Raspberry Pi Zero W. Segundo Multilógica Shop (2017), a voltagem operacional do módulo é de 3 a 5V, ele acompanha um LED indicador de estado e sua velocidade de transferência é de 9600 bits por segundo (bps). O autor ainda afirma que ele possui uma antena cerâmica.

Figura 25 - Módulo GPS Ubox GY-GPS6MV2



Fonte: Deal Extreme (2017, p.1).

3.2.2 Técnicas e ferramentas utilizadas

A construção da aplicação foi separada em duas partes: software embarcado e aplicativo *mobile*. Em ambas foi utilizada a ferramenta Visual Studio Code para editar o código fonte. O aplicativo móvel foi desenvolvido com a linguagem de programação Typescript e o sistema embarcado foi programado com Python.

Para o desenvolvimento do aplicativo móvel foi utilizada a *framework* NPM para o gerenciamento de dependências. A *framework* Ionic foi utilizada para construir e empacotar os elementos *front-end*, além de gerar um arquivo instalável para os sistemas operacionais Android e IOS. A biblioteca AngularJS foi utilizada para abstrair o código Typescript, nela foi possível utilizar diretivas para facilitar a interação entre o modelo e as páginas HyperText Markup Language (HTML). O AngularJS também foi utilizado para realizar as chamadas Hypertext Transfer Protocol (HTTP) para o sistema embarcado utilizando JSON. Foram adicionados módulos extras como: AgmCoreModule – para desenhar a localização em um mapa do Google Maps, IonicStorageModule – para persistir dados de localização, fotos e configurações, GaugesModule – para desenhar graficamente os medidores de temperatura, rotação e velocidade.

Para o desenvolvimento do sistema embarcado foi utilizada a linguagem de programação Python na sua versão 2.7 já disponibilizada no sistema operacional Rasppian. Foi necessária a biblioteca Flask que permite a criação de serviços HTTP. Foram utilizadas bibliotecas externas instaladas através do gerenciados de pacotes Python denominado PIP, o gerenciador de aplicações apt-get do SO Raspian e também, bibliotecas baixadas do site GitHub. A relação e instalação de todas as bibliotecas externas estão no Apêndice B.

3.2.3 Simuladores de central automotiva

Foi necessário o uso de simuladores para auxiliar no desenvolvimento da aplicação na parte da leitura de dados da central automotiva (ECU). Estes foram utilizados para facilitar o desenvolvimento do sistema embarcado na leitura da porta OBD2. Os simuladores são: OBDSim e ECU Simulator Pro.

Segundo Obdsim (2017), o simulador OBSim (ilustrado na Figura 26) foi desenvolvido da necessidade de simular dados de uma central automotiva para o software denominado OBD GPS Logger. Para a sua instalação no sistema operacional Windows, basta baixar um arquivo compactado do site do autor e rodar o arquivo executável obdsim.exe. O simulador possui as seguintes características:

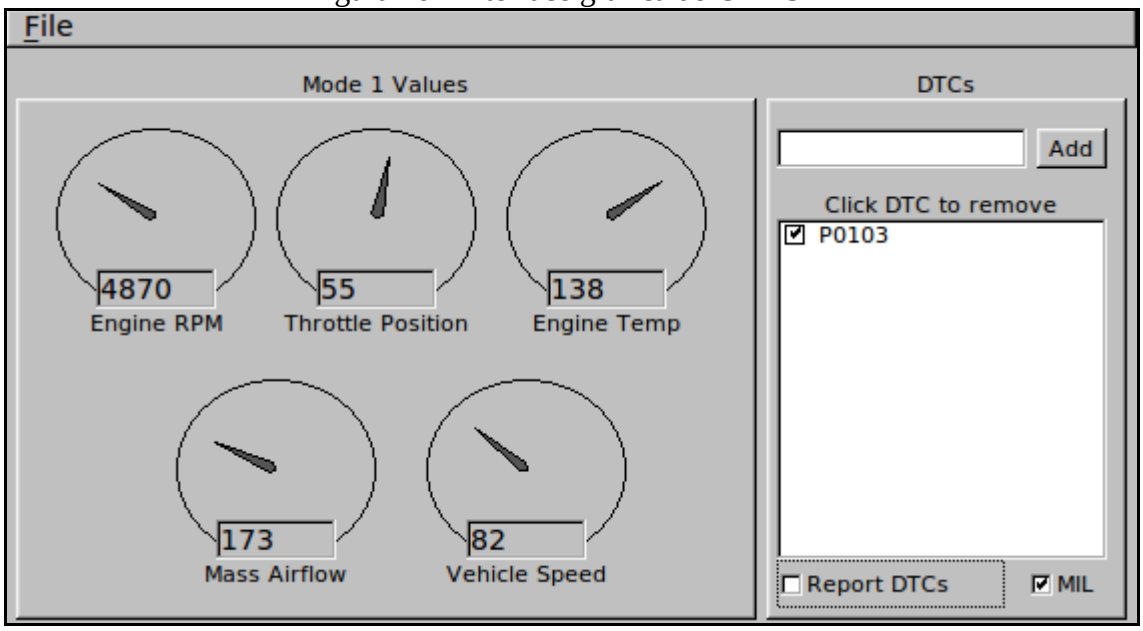
a) é executado pela linha de comando;

b) suporta comandos AT;

c) suporta múltiplos protocolos;

d) simula códigos de falha DTC.

Figura 26 - Interface gráfica do OBDSim



Fonte: OBDSim (2017, p.1).

Para utilizar o OBDSim de modo à pareá-lo com um dispositivo Bluetooth, é necessário abri-lo na linha de comando. É informado como argumento a porta de comunicação serial que o dispositivo Bluetooth está utilizando, como podemos observar no exemplo do Quadro 8.

Quadro 8 – Exemplo de comando para abrir o OBDSim

```
obdsim.exe -w COM10
```

Fonte: OBDSim (2017, p.1).

Já o simulador ECU Engine Pro (apresentado na Figura 27) é um aplicativo baixado através da loja Google Play e instalado no sistema operacional Android. Conforme Quoc (2017), este aplicativo converte um Android em adaptador ELM32 Bluetooth conectado à porta OBD2. O simulador possui as seguintes características:

a) suporta comandos AT;

b) suporta diversos PIDs;

c) suporta códigos de erros DTC;

d) utiliza o protocolo OBD2 ISO 15765-4 CAN.

Figura 27 - Interface do ECU Engine Pro

Fonte: Quoc (2017, p. 1).

Ambos os simuladores auxiliaram na leitura dos códigos DTC, na limpeza dos mesmo e também na leitura dos sensores da ECU.

3.2.4 Código fonte do sistema embarcado

Nesta seção são apresentados os principais códigos fonte implementados para o sistema embarcado e suas respectivas explicações. Os assuntos estão separados em subseções para separar o contexto de cada temática.

3.2.4.1 LEDs e botões

Para a utilização de LEDs e botões no sistema embarcado, foi necessária a biblioteca `RPi.GPIO`. Com ela, é possível fazer a configuração inicial dos pinos utilizados no barramento para, em seguida, realizar a sua utilização. O Quadro 9 demonstra configuração inicial de GPIO ilustrada na linha 12, esta é feita através do método `setup`. Para facilitar a utilização da classe `GPIOControl`, foi necessária a enumeração `Led` que contém os pinos responsáveis por cada LED, ela é descrita na linha 1.

Quadro 9 – Configuração inicial dos pinos do barramento

```
1   class Led:
2       BRANCO_1 = 29
3       BRANCO_2 = 37
4       VERDE = 33
5       AZUL = 35
6       VERMELHO = 31
7
8   class GPIOControl:
9       PIN_BOTAO = 40
10      def __init__(self):
11          self.setup()
12      def setup(self):
13          GPIO.setmode(GPIO.BOARD)
14          GPIO.setup(Led.BRANCO_1, GPIO.OUT)
15          GPIO.setup(Led.BRANCO_2, GPIO.OUT)
16          GPIO.setup(Led.VERDE, GPIO.OUT)
17          GPIO.setup(Led.AZUL, GPIO.OUT)
18          GPIO.setup(Led.VERMELHO, GPIO.OUT)
19          GPIO.setup(self.PIN_BOTAO, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
```

Fonte: elaborado pelo autor.

Após a configuração dos LEDs, a sua utilização é feita através de *threads*. Para facilitar, foi criada uma classe denominada `PiscaLedThread` (demonstrada no Quadro 10). Ela é necessária para piscar um LED por um determinado tempo opcionalmente informado. Também é possível informar se, após piscar, o LED permanecerá aceso. Todas essas configurações podem ser visualizadas na linha 2. Após, quando inicia-se a *thread*, é executado o método `run` (descrito na linha 9), este, é responsável por piscar o LED informado enquanto a variável `stopped` esteja com o valor `False`.

Quadro 10 - Criação da classe `PiscaLedThread`

```
1   class PiscaLedThread(threading.Thread):
2       def __init__(self, gpio_control, led, tempo=0.2, aceso=True):
3           self.stopped = False
4           self.gpio_control = gpio_control
5           self.led = led
6           self.tempo = tempo
7           self.aceso = aceso
8           threading.Thread.__init__(self)
9       def run(self):
10          while not self.stopped:
11              GPIO.output(self.led, GPIO.HIGH)  # led on
12              time.sleep(self.tempo)
13              GPIO.output(self.led, GPIO.LOW)  # led off
14              time.sleep(self.tempo)
15          if self.aceso:
16                  GPIO.output(self.led, GPIO.HIGH)  # led on
17      def stop(self):
18          self.stopped = True
```

Fonte: elaborado pelo autor.

No Quadro 11 é apresentado o trecho de código que representa o método `run` da *thread* responsável por aguardar o botão de *reset* ser pressionado por três segundos. Na linha 5, verifica-se o botão para em seguida calcular-se o tempo. O tempo do botão pressionado é

calculado na linha 10 e armazenado na variável `intervalo`, se o botão for pressionado por mais de três segundos, apagam-se todos os leds e executa a função `mainFn`, esta função reinicializa o sistema embarcado.

Quadro 11 - *Thread* para verificar botão reset

```
1   def run(self):
2       newDate = None
3       intervalo = 0
4       while not self.stopped:
5           if GPIO.input(self.gpio_control.PIN_BOTAO) == 1:
6               if not self.pressed:
7                   newDate = datetime.now() + timedelta(seconds=3)
8                   self.pressed = True
9               else:
10                  intervalo = (newDate - datetime.now()).total_seconds()
11                  if intervalo <= 0:
12                      print("HARDWARE RESETED BY USER...")
13                      self.gpio_control.apaga_todos_leds()
14                      self.mainFn(self.opt)
15                      self.stop()
16          else:
17              self.pressed = False
```

Fonte: elaborado pelo autor.

No Quadro 12 é descrito o trecho de código executado ao iniciar o Raspberry Pi Zero W. Na linha 11, o código indica que quando o botão cuja GPIO 3 (pino 05) for pressionado, irá chamar a função `shutdown` (apresentada na linha 8). Esta função executa uma comando na linha 9 que desliga do sistema operacional Raspian.

Quadro 12 - Código do botão desligar/ligar o SO

```
1   import RPi.GPIO as GPIO
2   import time
3   import os
4
5   GPIO.setmode(GPIO.BCM)
6   GPIO.setup(3, GPIO.IN)
7
8   def shutdown(channel):
9       os.system("sudo shutdown -h now")
10
11  GPIO.add_event_detect(3, GPIO.FALLING, callback=shutdown)
12  while True:
13      time.sleep(1)
```

Fonte: elaborado pelo autor.

3.2.4.2 Conexão de rede 3G e DDNS

Para a conexão com Internet foi utilizada a rede 3G em conjunto com o serviço de DNS dinâmico (DDNS) denominado No-IP. Para realizar a conexão 3G foi necessária a ferramenta `wvdial` que é responsável pela discagem e obtenção do IP através do modem 3G. A `wvdial` utiliza uma configuração de Access Point Name (APN) chamada `Dialer tim` para

os dados de conexão da operadora de telefonia móvel TIM. Esta é descrita no Quadro 13 e permite que a ferramenta identifique quais os dados de discagem deve utilizar.

Quadro 13 - Configuração de APN da operadora TIM

```
 GNU nano 2.2.6     Arquivo: wvdial.conf

[Dialer tim]
Init2 = ATZ
Init3 = AT+CGDCONT=1,"ip","tim.br"
Stupid Mode = 1
ISDN = 0
Phone = *99***1#
Ask Password = 0
Modem = /dev/gsmmodem
Username = tim
Dial Command = ATD
Password = tim
Baud = 460800
```

Fonte: elaborado pelo autor.

No Quadro 14 é exposto o trecho de código para a obtenção do IP e a configuração do No-IP. A linha 1 e 2 indica que o primeiro LED branco começa a piscar para informar que está executando este procedimento de configuração IP. Observa-se que na linha 4 força-se o término da aplicação `wvdial` para, em seguida, na linha 6 iniciar-se novamente utilizando a configuração de APN `tim`. Na linha 8, é armazenado o IP obtido através da conexão 3G na variável `ipPpp0`. Esta variável é verificada em seguida e caso obtiver um IP público (maior que `100`), é executada a configuração do No-IP, se não, pisca-se o LED vermelho indicando que este processo falhou e inicia-se novamente essa configuração.

Após a obtenção do IP público, é configurado o No-IP. Esta configuração é demonstrada na linha 20 e 22. Na linha 20, obriga-se a parada do processo `noip2`, para na linha 22 iniciar novamente o processo. Por fim, na linha 24 a *thread* responsável por piscar o LED é finalizada e o primeiro LED branco permanece aceso para informar sucesso na execução de configuração IP.

Quadro 14 – Obtenção de IP e configuração do No-IP

```
1   thread_leds = PiscaLedThread(self.gpio_control, Led.BRANCO_1)
2   thread_leds.start()
3   while True:
4       os.popen("sudo pkill -9 -f wvdial").read()
5       time.sleep(1)
6       os.popen("sudo wvdial tim &")
7       time.sleep(7)
8       ipPpp0 = str(os.popen("ifconfig ppp0 | grep inet")
9           .read())
10          .replace("          inet end.: ", "")[:3]
11      if int(ipPpp0) <= 100:
12          self.gpio_control.pisca_led(Led.VERMELHO, tempo=1, aceso=False)
13          continue
14      break
15
16  thread_leds.stop()
17  thread_leds = PiscaLedThread(self.gpio_control, Led.BRANCO_1, tempo=0.05)
18  thread_leds.start()
19
20  os.popen("sudo pkill -9 -f noip2").read()
21  time.sleep(2)
22  os.popen("sudo noip2").read()
23
24  thread_leds.stop()
```

Fonte: elaborado pelo autor.

3.2.4.3 Bluetooth

O Quadro 15 apresenta o código fonte responsável pela configuração do Bluetooth. A primeira ação do método na linha 2 é executar o comando para liberar a porta `RFCOMM 0`, para em seguida, configurar a sua utilização. Este código recebe opcionalmente por parâmetro um endereço Media Access Control (MAC) que caso informado, na linha 4 ele é passado ao método responsável por recuperar a porta do OBD2, para em seguida executar um comando `bind` que irá fazer com que o sistema operacional identifique o MAC e a porta Bluetooth responsável pelo OBD2.

Se o MAC não for informado, na linha 9 são pesquisados dispositivos Bluetooth durante o período de 10 segundos. Para cada dispositivo encontrado, na linha 12 procura-se a porta responsável pelo OBD2. Caso a porta OBD2 existir, é executado um `bind` na porta 0 com a porta e o MAC.

Quadro 15 - Método para configuração do Bluetooth

```
1   def configurar_bluetooth(self, mac_addr=None):
2       os.popen('sudo rfcomm release 0').read()
3       if mac_addr is not None:
4           port = self.bt_recuperar_servico_obd(mac_addr)
5           if port is not None:
6               os.system("sudo rfcomm bind 0 %s %s " % (addr, port))
7               return True
8           return False
9       nearby_devices = bluetooth.discover_devices(
10          duration=10, lookup_names=True, flush_cache=True, lookup_class=False)
11      for addr, name in nearby_devices:
12          port = self.bt_recuperar_servico_obd(addr)
13          if port is not None:
14              os.system("sudo rfcomm bind 0 %s %s " % (addr, port))
15              return True
16      return False
```

Fonte: elaborado pelo autor.

No Quadro 16 é descrito o método para recuperar a porta de execução do OBD2 à partir dos serviços disponíveis no endereço (variável `addr`) informado por parâmetro. Na linha 2 são pesquisados os serviços para em seguida, na linha 6 verificar se o protocolo é `RFCOMM` e por fim, se o nome do serviço contiver em sua composição os caracteres “COM”, for igual A “SPP” ou for igual a “BLT”, é retornado neste método o número da porta do serviço. Caso contrário este método retorna `None` indicando que não foi encontrado nenhum serviço OBD2.

Quadro 16 - Método para recuperar a porta do serviço OBD2

```
1   def bt_recuperar_servico_obd(self, addr):
2       services = bluetooth.find_service(address=addr)
3       port = None
4       if len(services) > 0:
5           for svc in services:
6               if(svc["protocol"] == "RFCOMM"):
7                   if("COM" in svc["name"] or
8                       svc["name"] == "SPP" or
9                       svc["name"] == "BLT"):
10                      return int(svc["port"])
11      return None
```

Fonte: elaborado pelo autor.

3.2.4.4 Armazenamento de configurações

A aplicação embarcada utiliza configurações salvas em arquivo texto com seu conteúdo no formato JSON (ilustrado no Quadro 17). As informações armazenadas são referentes ao dados do usuário como e-mail e celular, bem como permissões de notificação. Por fim, o JSON armazena o simulador utilizado, este valor é de 0 à 2 onde:

a) 0: representa que não está utilizando nenhum simulador;

b) 1: identifica o uso do simulador OBDSim;

c) 2: sinaliza a utilização do simulador ECU Engine Pro.

Quadro 17 - Formato do JSON salvo no arquivo texto

```
{
   "celular":"+5547992929968",
   "email":"maicon.gerardi@gmail.com",
   "notificarEmail":true,
   "notificarSMS":true,
   "simulador":"1"
}
```

Fonte: elaborado pelo autor.

Essas configurações são salvas conforme o Quadro 18, onde, na linha 4 é enviado o JSON no corpo da requisição HTTP POST recuperado e armazenado na variável `content`. A linha 5 e 6 respectivamente são utilizadas para armazenar o JSON no arquivo `configs.txt`.

Quadro 18 - Método para salvar as configurações da aplicação

```
1  @app.route('/save_configs', methods=['POST'])
2  def save_configs():
3      try:
4          content = request.get_json()
5          with io.open('./database/configs.txt', 'w', encoding='utf-8') as f:
6              f.write(json.dumps(content, ensure_ascii=False))
7          return json.dumps(dict(status =  "OK"))
8      except Exception as ex:
9          return json.dumps(dict(error = str(ex)))
```

Fonte: elaborado pelo autor.

No Quadro 19 é demonstrado a recuperação da configuração armazenada. Na linha 4 e 5 respectivamente é aberto o arquivo `configs.txt` e é armazenado o seu conteúdo na variável `content` para, na linha 7 transformar este conteúdo em um objeto Python onde, o mesmo é retornado na linha 14.

Quadro 19 - Recuperação das configurações

```
1  def get_configs():
2      configs = None
3      try:
4          with io.open('./database/configs.txt', 'r') as f:
5              content = f.read()
6          if content is not None and content:
7              configs = json.loads(content,
8                  object_hook=lambda d: namedtuple('X', d.keys())(*d.values()))
9      except IOError as ex:
10         if "No such file or directory" in str(ex):
11             configs = None
12         else:
13             raise ex
14     return configs
```

Fonte: elaborado pelo autor.

3.2.4.5 Câmera

No Quadro 20 é demonstrado o código fonte responsável pela recuperação de uma foto do módulo Raspberry Pi Camera. Foi utilizada a biblioteca `flask-video-streaming` (importada na linha 1) que disponibiliza a classe `Camera` para manipular fotos e disponibilizar

o *streamming* da câmera em tempo real. Na linha 3 é instanciado um objeto do tipo `Camera` armazenado na variável `my_camera` que será utilizado por todo o sistema embarcado. O método da linha 6 denominado `get_foto` disponibiliza uma foto extraída da câmera. Na linha 8 é retornado uma cadeia de caracteres contendo o cabeçalho de uma imagem em extensão .jpeg com a captura de um frame da câmera codificado em base64[2].

Quadro 20 - Código fonte para recuperar uma foto da câmera

```
1  from video_streaming.camera_pi import Camera
2
3  my_camera = Camera()
4
5  @app.route('/get_foto')
6  def get_foto():
7      global my_camera
8      return '' + base64.b64encode(my_camera.get_frame())
```

Fonte: elaborado pelo autor.

O Quadro 21 disponibiliza o código fonte para recuperar o *streamming* da câmera em tempo real. Este *streamming* é capturado e retornando na linha 4. O objeto `Response` retorna quadros (*frames*) da câmera chamando o método `gen`. A linha 7 apresenta o método `gen`, este é responsável por retornar os *frames* da câmera quando solicitados. Na linha 9 é capturado o quadro e armazenado na variável `frame` que em seguida é retornado na linha 11.

Quadro 21 - Recuperar *streamming* da câmera

```
1  @app.route('/get_video')
2  def get_video():
3      global my_camera
4      return Response(gen(my_camera),
5                      mimetype='multipart/x-mixed-replace; boundary=frame')
6
7  def gen(camera):
8      while True:
9          frame = camera.get_frame()
10         yield (b'--frame\r\n'
11             b'Content-Type: image/jpeg\r\n\r\n' + frame + b'\r\n')
```

Fonte: elaborado pelo autor.

No Quadro 22 encontra-se o código fonte responsável por capturar *frames* da câmera. Ele foi adaptado da biblioteca `flask-video-streaming`, a sua adaptação foi feita na linha 5 cuja a resolução da câmera foi diminuída para aumentar a velocidade de transmissão das imagens em tempo real. O código captura imagens da câmera na linha 6 para em seguida disponibilizá-las na linha 11. Isso acontece até que a aplicação embarcada finalize.

[2] A codificação Base64 consiste em utilizar carácteres US-ASCII (não acentuados) para codificar qualquer tipo de dados em 8 bits (CCM, 2017, p. 1).

Quadro 22 - Código que retorna os *frames* da câmera

```
1   def get_frames():
2       with picamera.PiCamera() as camera:
3           time.sleep(2)
4           stream = io.BytesIO()
5           camera.resolution = (160, 120)
6           for foo in camera.capture_continuous(stream, 'jpeg',
7                                                use_video_port=True):
8               if BaseCamera.stopped:
9                   break
10              stream.seek(0)
11              yield stream.read()
12              stream.seek(0)
13              stream.truncate()
```

Fonte: adaptado de Grinberg (2014, p. 1).

3.2.4.6 GPS

No Quadro 23 é apresentado o código fonte responsável por recuperar os dados do módulo GPS. Observa-se na linha 3 que o módulo é disponibilizado através de uma comunicação serial no caminho `/dev/serial0` e sua taxa de transferência é configurada para 9600 bps, a comunicação é disponibilizada em uma variável chamada `port`. Segundo Baumgarten (2016), é necessário fazer a mineração do retorno para capturar a longitude e latitude. O padrão NMEA basicamente retorna 4 sentenças, sendo elas `$GPGGA`, `$GPGSA`, `$GPGSV` e `$GPRMC`. A latitude e longitude são encontradas na sentença `$GPGGA` e a linha 14 busca por ela à cada nova linha recuperada pela comunicação serial.

Após a recuperação da sentença `$GPGGA`, entre a linha 15 e 24 são recuperados os dados de latitude e longitude. Na linha 15, os valores da sentença são separados por “,” e armazenados no vetor de string denominado `retorno`. Em seguida, na linha 16 é verificado se o vetor `retorno` na posição 2 possui valor, se for vazio significa que o módulo está sem sinal, portanto retorna uma mensagem ao usuário conforme linha 17. Se houver valor na sentença, este é retornado na linha 24, nota-se que é feita uma conversão de valores no formato de Degrees Minutes Seconds (DMS) para o formato Decimal Degrees (DD).

Quadro 23 - Recuperar dados do módulo GPS

```
1  @app.route('/get_gps')
2  def get_gps():
3      port = serial.Serial("/dev/serial0", baudrate=9600, timeout=10.0)
4      line = []
5      count = 1
6      while count < 15:
7          try:
8              rcv = port.read()
9          except:
10             rcv = ''
11         line.append(rcv)
12         if rcv == '\n':
13             line = "".join(line)
14             if line.find("GPGGA") != -1:
15                 retorno = line.split(',')
16                 if retorno[2] == '':
17                     return json.dumps(dict(error =  "Sem Sinal"))
18                 lat = int(retorno[2][0:2]) + (float(retorno[2][2:len(retorno[2])]) / 60)
19                 if retorno[3] == 'S' :
20                     lat = lat * -1
21                 longit = int(retorno[4][0:3]) + (float(retorno[4][3:len(retorno[4])]) / 60)
22                 if retorno[5] == 'W':
23                     longit = longit * -1
24                 return json.dumps(dict(lat=lat, longit=longit))
25             line = []
26             count = count + 1
27     return json.dumps(dict(error =  "Sem Dados na porta /dev/serial0"))
```

Fonte: adaptado de Baumgarten (2016, p. 35).

3.2.4.7 OBD

Para organizar melhor a interação com os sensores e a leitura do adaptador ELM327 de modo geral, foi criada uma classe chamada `OBDControl`. Esta classe possui os métodos para conectar com o adaptador ELM327, recuperar os pids suportados pelo veículo, recuperar valores de pids e também recuperar informações dos códigos de erro (DTC).

No Quadro 24 é apresentado a maneira para obter a conexão OBD, está é feita através da biblioteca `python-obd` pela classe `OBD`. Na linha 3 é configurada e obtida esta conexão e armazenada na variável `_connection` da classe `OBDControl`, esta variável irá ser utilizada pelo sistema embarcado para recuperar valores de PIDs e também códigos de erro (DTC). Caso não seja possível conectar com o adaptador ELM327, na linha 5 é disparada uma exceção informando que não foi conectado. Caso contrário, na linha 6 é retornada a conexão OBD.

Quadro 24 - Obtenção da conexão OBD

```
1  def _connect_obd(self):
2      try:
3          self._connection = obd.OBD(baudrate=9600,portstr='/dev/rfcomm0',fast=False)
4          if  self._connection.status() == OBDStatus.NOT_CONNECTED:
5              raise Exception("nao conectado com ELM327.")
6          return self._connection
7      except Exception as ex:
8          self._connection = None
9          raise Exception(ex)
```

Fonte: elaborado pelo autor.

A técnica de sincronização de *threads* é utilizada no sistema embarcado para evitar que dois processos utilizem a mesma conexão serial com o dispositivo ELM327 ao mesmo tempo. Esta sincronização é feita através de bloqueios denominados *locks* que são disponibilizados nativamente pela linguagem Python no módulo `threading`. O Quadro 25 apresenta esta dessa técnica, na linha 1 é criado um objeto `Lock` que é utilizado na linha 4. Este lock impede que duas *threads* acessem o bloco `with` (linha 4) simultaneamente. Isso evita que as duas *threads* obtenham acesso à conexão OBD2 ao mesmo tempo.

Quadro 25 – Lock utilizado para obter conexão OBD2

```
1   _lock_connection = threading.Lock()
2
3   def get_connection(self):
4    with self._lock_connection:
5        if self._connection is None:
6            return self._connect_obd()
7        elif self._connection.status() == OBDStatus.NOT_CONNECTED:
8            return self._connect_obd()
9        else:
10           return self._connection
```

Fonte: elaborado pelo autor.

3.2.4.7.1 Leitura de PIDs

No Quadro 26 é apresentado o código fonte responsável pela recuperação de PIDs suportados pela ECU. Ele é utilizado quando a classe `OBDControl` é inicializada ou quando o sistema embarcado solicita a leitura dos PIDs. Na linha 2 verifica-se a variável de classe `_supported_pids` não é nula (`None`), se ela não for nula significa que os PIDs suportados já foram verificados, portanto, retorna o valor da variável para o método. Caso contrário, na linha 4 é recuperada a conexão OBD para em seguida, na linha 6 verificar os pids suportados. Após esse procedimento, itera-se sobre a lista de comandos suportados para, na linha 11, fazer o retorno desses PIDs. Nota-se que na linha 7 são ignorados os comandos de leitura dos códigos DTC e também os comandos de controle.

Quadro 26 - Recuperar PIDs suportados pela ECU

```
1   def get_supported_pids(self):
2       if self._supported_pids is not None:
3           return self._supported_pids
4       connection = self.get_connection()
5       self._supported_pids = []
6       for command in connection.get_supported_commands():
7           if command.command not in [ b"03", b"07", b"04", b"0100", b"0120",
8                                       b"0140", b"0600", b"0620", b"0640",
9                                       b"0660", b"0680", b"06A0", b"0101" ]:
10              self._supported_pids.append(command)
11      return self._supported_pids
```

Fonte: elaborado pelo autor.

O Quadro 27 ilustra o método `execute_query` da classe `OBDControl`. Ele é utilizado na execução de comandos para a central automotiva (ECU) através do adaptador ELM327. Na linha 2 é obtido a conexão para, em seguida, na linha 3 executar o respectivo comando informado por parâmetro. O retorno do método depende do tipo de comando informado (variável `cmd`).

Quadro 27 – Execução de comandos OBD

```
1   def execute_query(self, cmd):
2       connection = self.get_connection()
3       return self._connection.query(cmd)
```

Fonte: elaborado pelo autor.

No Quadro 28 é enumerado o código fonte responsável pela obtenção de valores dos PIDs suportados pela central automotiva (ECU). Observa-se que na linha 5 é criada uma lista denominada `listSensors` que irá armazenar dicionários com o código do PID e o seu respectivo valor. Após essa criação, na linha 6 é verificado os PIDs suportados, e para cada um é executada a sua respectiva *query* que irá retornar o valor do PID. Da linha 10 até a linha 20 é feito o tratamento de dados, que arredondam os valores decimais para 4 casas. Na linha 21 é adicionado esta leitura na `listSensors`. Já na linha 23 é retornado todos os valores dos PIDs em formato JSON ordenados pelos seus respectivos códigos.

Quadro 28 - Obtenção de valores dos PIDs

```
1   @app.route('/get_obdii_values')
2   def get_obdii_values():
3       global obd_control
4       try:
5           listSensors = []
6           for command in obd_control.get_supported_pids():
7               cmd = command
8               response = obd_control.execute_query(cmd)
9               valor = ""
10              try:
11                  valor = response.value.magnitude
12                  if isinstance(valor, numbers.Real):
13                      if valor % 1 == 0:
14                          valor = int(valor)
15                      else:
16                          valor = str(float(str("%.4f" % round(valor,4))))
17                  else:
18                      valor = str(valor)
19              except AttributeError:
20                  valor = response.value
21              listSensors.append(dict(codigo=str(cmd.command),
22                                      valor=str(valor)))
23          return json.dumps(sorted(listSensors, key=lambda s: s.get('codigo')))
24      except Exception as ex:
25          return json.dumps(dict(error = str(ex)))
```

Fonte: elaborado pelo autor.

3.2.4.7.2 Leitura de códigos de erro DTC

Para a leitura dos códigos de erro DTC foram utilizados os serviços 0x03 e 0x07 que retornam respectivamente os códigos pendentes e registrados. Para a sua obtenção foi modificado o código fonte original da biblioteca `python-obd` para suportar os dois simuladores utilizados que são o OBDSim e ECU Engine Pro, além de suportar a leitura sem um simulador definido. O código fonte dessa modificação está descrito no Quadro 29.

O parâmetro `simulador` informado na linha 1 identifica qual é o simulador que será utilizado para a leitura dos códigos DTC, os valores são: 0 – nenhum simulador, 1 – OBDSim, 2 – ECU Engine Pro. Na linha 3 é executado o método padrão da biblioteca caso não necessite simulador. Da linha 13 até a linha 29 são executados passos para converter os valores hexadecimais para códigos DTC de acordo com cada simulador. Por fim, são retornados todos os códigos DTC na linha 30. O `query_dtc` é utilizados na no método `get_status_dtc`, conforme o Quadro 30.

Quadro 29 - Código fonte para a leitura do serviços 0x03 e 0x07

```
1  def query_dtc(self, cmd=commands.GET_DTC, simulador=0):
2      if simulador == 0:
3          return self.query(cmd).value
4      if self.status() == OBDStatus.NOT_CONNECTED:
5          return OBDResponse()
6      if not self.test_cmd(cmd):
7          return OBDResponse()
8      cmd_string = self.__build_command_string(cmd)
9      lines = self.interface.send_and_parse(cmd=cmd_string, return_lines=True)
10     if cmd_string:
11         self.__last_command = cmd_string
12     codes = []
13     for line in lines:
14         line = line.replace(" ", "")
15         if "NO" in line:
16             continue
17         bytes_line = bytearray()
18         ## SE FOR O OBDSIM
19         if simulador == 1:
20             line = line[7:]
21             bytes_line = bytearray(unhexlify(line))
22         ## SE FOR O ECU Engine PRO
23         elif simulador == 2:
24             line = line[9:]
25             bytes_line = bytearray(unhexlify(line))
26         for n in range(1, len(bytes_line), 2):
27             dtc = parse_dtc( (bytes_line[n-1], bytes_line[n]) )
28             if dtc is not None:
29                 codes.append(dtc)
30     return codes
```

Fonte: adaptado de Python-obd (2017, p. 1).

O Quadro 30 trata de um método denominado `get_status_dtc` que é responsável por organizar as informações de códigos de erro DTC e também obtém o status da MIL. Na linha

2 é obtida a conexão para em seguida, nas linhas 4 e 5, fazer a recuperação dos códigos pendentes e registrados. Na linha 8 é recuperado o estado da lâmpada MIL. A linha 9 até a linha 22 serve para organizar as informações relevantes para o sistema embarcado e, na linha23, efetuar o retorno das mesmas.

Quadro 30 - Código fonte para organizar os DTC

```
1  def get_status_dtc(self):
2      connection = self.get_connection()
3      simulador = get_simulador()
4      dtc_registrados = connection.query_dtc(simulador=simulador)
5      dtc_pendentes = connection.query_dtc(cmd=obd.commands.GET_CURRENT_DTC,
6                                           simulador=simulador)
7      status = connection.query(obd.commands.STATUS)
8      status = status.value
9      _registrados = []
10     for codigo, descricao in dtc_registrados:
11         _registrados.append(dict(codigo=codigo,
12                                  descricao=descricao,
13                                  url='http://www.troublecodes.net/'+str(codigo)))
14     _pendentes = []
15     for codigo, descricao in dtc_pendentes:
16         _pendentes.append(dict(codigo=codigo,
17                                descricao=descricao,
18                                url='http://www.troublecodes.net/'+str(codigo)))
19     _status = None
20     if status is not None:
21         _status = Status(MIL=status.MIL,
22                          qtd_erros=status.DTC_count)
23     return StatusDTC(_pendentes,
24                      _registrados,
25                      _status)
```

Fonte: elaborado pelo autor.

No Quadro 31 são enumeradas duas classes: `Status` e `StatusDTC`. Elas são responsáveis por organizarem os dados de leitura dos códigos de erro DTC para retorná-los ao usuário.

Quadro 31 - Classe `Status` e classe `StatusDTC`

```
1   class Status:
2       def __init__(self, MIL, qtd_erros):
3           self.MIL = MIL
4           self.qtd_erros = qtd_erros
5
6       def json_dump(self):
7           return dict(MIL=self.MIL, qtd_erros=self.qtd_erros)
8
9   class StatusDTC:
10      def __init__(self, registrados, pendentes, status):
11          self.registrados = registrados
12          self.pendentes = pendentes
13          self.status = status
14
15      def json_dump(self):
16          status = None
17          if self.status is not None:
18              status = self.status.json_dump()
19          return dict(dtc_registrados=self.registrados,
20                      dtc_pendentes=self.pendentes,
21                      status=status)
```

Fonte: elaborado pelo autor.

No Quadro 32 é apresentado o código fonte referente à limpeza dos códigos de erro DTC. Para executar esta operação foi necessário enviar o comando `CLEAR_DTC` (descrito na linha 5). Após a execução neste comando, caso não houver nenhuma exceção gerada pelo comando, todos os códigos pendentes e registrados são apagados da ECU. Além disso, na linha 7 o método remove o arquivo `status.txt` que armazena os códigos DTC já capturados pela rotina de verificação de DTCs.

Quadro 32 - Limpeza dos códigos de erro DTC

```
1   @app.route('/clear_dtc')
2   def clear_dtc():
3       global obd_control
4       try:
5           obd_control.execute_query(obd.commands.CLEAR_DTC)
6           try:
7               os.remove("./database/status.txt")
8           except Exception as ex:
9               if not "No such file or directory" in str(ex):
10                  raise Exception(ex)
11          return json.dumps(dict(status =  "OK"))
12      except Exception as ex:
13          return json.dumps(dict(error = str(ex)))
```

Fonte: elaborado pelo autor.

3.2.4.7.3 Notificação de novos códigos DTC

A verificação e notificação dos DTC ocorrem de 20 em 20 segundos e é independe da interação do usuário para a sua inicialização. O processo é executado pelo servidor embarcado antes da sua disponibilização. Para facilitar a sua manipulação, foi codificada uma classe para abstrair a sua complexidade, esta classe denomina-se `DTCControl` e está exposta no Quadro

33. Na linha 7 é armazenado o intervalo de execução de cada processo na variável `delay`. Na linha 10 é declarado o método responsável por inicializar o monitor de códigos DTC. Este método é denominado `start_monitor` e, na linha 11, inicializa um temporizador que executa o método `monitorar_dtcs` (descrito no Quadro 34) no intervalo armazenado na variável `delay` (que neste caso é 20 segundos). Ou seja, após a execução da linha 12, o método `monitorar_dtcs` é disparado após 20 segundos.

Quadro 33 - Classe `DTCControl`

```
1   class DTCControl:
2       delay = 0
3       obd_control = None
4
5       def __init__(self, obd_control):
6           self.obd_control = obd_control
7           self.delay = 20
8           self.start_monitor()
9
10      def start_monitor(self):
11          t = Timer(self.delay, self.monitorar_dtcs)
12          t.start()
```

Fonte: elaborado pelo autor.

O método denominado `monitorar_dtcs` exposto no Quadro 34 é utilizado para fazer o monitoramento dos códigos DTC. Na linha 3 as configurações são recuperadas e armazenadas na variável `configs` e caso elas não existam, o método dispara uma exceção informado que está sem configurações. Na linha 6 são recuperados os DTC no momento atual da leitura e recuperam-se, posteriormente na linha 7, os códigos DTC já notificados pela aplicação. Da linha 11 até a linha 18 são mesclados os códigos já notificados com os códigos lidos atualmente para, na linha 19 restar somente os novos códigos pendentes de notificação.

Na linha 22 é enviado uma notificação de SMS e na linha 26 é enviada uma notificação de e-mail ao usuário. Essa notificação informa que existem códigos de erro encontrados no veículo com os seus respectivos códigos de erro. Observa-se na linha 21 e 24 que ela só ocorre se as *flags* de notificação `notificarSMS` e `notificarEmail` estiverem como `True`. Além disso, na linha 27, os códigos recuperados acrescentados na variável `db_status` para ser, em seguida na linha 28, chamar o método responsável por salvar esses códigos DTC. Por fim, na linha 30 é chamado o método `start_monitor`, que vai novamente inicializar um temporizador para executar o método `monitorar_dtcs`. O monitor só é encerrado caso o sistema embarcado for finalizado.

Quadro 34 - Método para monitoramento de DTC

```
1   def monitorar_dtcs(self):
2       try:
3           configs = get_configs()
4           if configs is None:
5               raise Exception("Sem configuracoes...")
6           dtcs = self.obd_control.get_status_dtc()
7           db_status = self.get_db_status()
8           if db_status is None:
9               db_status = []
10          _novos_dtcs = []
11          for dtc in dtcs.registrados:
12              codigo = dtc.get('codigo')
13              if codigo not in db_status:
14                  _novos_dtcs.append(codigo)
15          for dtc in dtcs.pendentes:
16              codigo = dtc.get('codigo')
17              if codigo not in db_status:
18                  _novos_dtcs.append(codigo)
19          _novos_dtcs = list(set(_novos_dtcs))
20          if len(_novos_dtcs) > 0 and
21              if configs.notificarSMS:
22                  send_sms(configs.celular,
23                           'Codigos de erro encontrados em seu veiculo! ' + str(_novos_dtcs))
24              if configs.notificarEmail:
25                  send_email(configs.email,
26                           'Codigos de erro encontrados em seu veiculo! ' + str(_novos_dtcs))
27              db_status += _novos_dtcs
28              self.save_db_status(db_status)
29      finally:
30          self.start_monitor()
```

Fonte: elaborado pelo autor.

No Quadro 35 são expostos dois métodos responsável por salvar e recuperar os códigos DTC já notificados ao usuário. Isso é utilizado para que a notificação não se repita mais de uma vez até que execute a limpeza dos códigos. Na linha 1 é declarado o método para recuperar os DTCs, este método chama-se `get_db_status` e na linha 4 abre-se o arquivo `status.txt`[3] para, na linha 7 transformar o seu conteúdo em um objeto Python e retornar em seguida. Já na linha 16 é declarado o método responsável por salvar os códigos DTC já notificados ao usuário, este método é denominado `save_db_status` e recebe como parâmetro os códigos DTC na variável `status`. Por fim, na linha 19 é persistida a variável `status`.

[3] O arquivo `status.txt` armazena um vetor de strings que representam os códigos DTC já notificados ao usuário.

Quadro 35 - Métodos para salvar DTCs

```
1   def get_db_status(self):
2       status = None
3       try:
4           with io.open('./database/status.txt', 'r') as f:
5               content = f.read()
6           if content is not None and content:
7               status = json.loads(content,
8                                   object_hook=lambda d: namedtuple('X', d.keys())(*d.values()))
9       except IOError as ex:
10          if "No such file or directory" in str(ex):
11              status = None
12          else:
13              raise ex
14      return status
15
16  def save_db_status(self, status):
17      try:
18          with io.open('./database/status.txt', 'w', encoding='utf-8') as f:
19              f.write(unicode(json.dumps(status, ensure_ascii=False)))
20          return dict(status =  "OK")
21      except Exception as ex:
22          return dict(error = str(ex))
```

Fonte: elaborado pelo autor.

No Quadro 36 é exposta a função responsável por enviar SMS. Isto é feito através da biblioteca `twilio` e é necessário informar um SID e TOKEN previamente obtidos gratuitamente no site da biblioteca através de um cadastro realizado no próprio site. Na linha 5 é enviada a mensagem através da classe `Client` da biblioteca onde, nela é informada o número de destino (linha 6) e a mensagem encaminhada (linha 8).

Quadro 36 - Função para enviar SMS

```
1   def send_sms(numero, mensagem):
2       account_sid = "SID"
3       auth_token  = "TOKEN"
4       client = Client(account_sid, auth_token)
5       message = client.messages.create(
6           to=numero,
7           from_="NUMERO",
8           body=mensagem)
```

Fonte: elaborado pelo autor.

No Quadro 37 é demonstrado o código fonte responsável pelo envio de e-mail, este utiliza a biblioteca nativa `smtplib` que disponibiliza os métodos necessários. Na linha 2 é configurado o Simple Mail Transfer Protocol (SMTP) de envio para em seguida, na linha 6 efetuar-se a autenticação utilizando o usuário e senha do servidor. A mensagem e o endereço de e-mail de destino são passados como parâmetro na linha 1 para esta função, na linha 8, efetuar o seu envio.

Quadro 37 - Função para envio de e-mail

```
1  def send_email(email, mensagem):
2      server = smtplib.SMTP('smtp.gmail.com', 587)
3      server.ehlo()
4      server.starttls()
5      server.ehlo()
6      server.login("NOME_DE_USUARIO", "SENHA")
7      msg = 'Subject: {}\n\n{}'.format('[Monitor Do Veiculo] IMPORTANTE!', mensagem)
8      server.sendmail("monitor.tcc2@gmail.com", email, msg)
```

Fonte: elaborado pelo autor.

3.2.5 Código fonte da aplicação *mobile*

Nesta seção são descritos os principais código fontes implementados para a camada *front-end* da aplicação *mobile*. Os assuntos estão separados em subseções para dividir o contexto de cada temática.

3.2.5.1 Chamadas HTTP para o servidor embarcado

A aplicação móvel desenvolvida executa chamadas HTTP para o servidor embarcado, as chamadas são encapsuladas em métodos de uma classe denominada `HttpSerice`. Os métodos são `get` e `post`. No Quadro 38 é possível visualizar o código fonte do método `get`, este, é responsável por executar chamadas do tipo GET ao servidor embarcado. Nota-se que na linha 3 é utilizado o objeto `Http` da biblioteca `AngularJS`. Na linha 17 é configurado um tempo para aguardar a chamada do serviço e caso essa chamada não seja executada no tempo definido na variável estática `DEFAULT_TIMEOUT` da classe `AppSettings`, é lançada uma exceção informando um *timeout*. Se a chamada for executada com sucesso, é retornado um objeto do tipo `Promise` contendo a resposta da requisição no objeto `Response`.

Quadro 38 - Método HTTP para chamadas do tipo GET

```
1  public get(url: string, headersDict = null) {
2      return new Observable<any>(observer => {
3          this.http
4              .get(`${url}`, {headers: new Headers(headersDict)}).subscribe(data =>{
5                  observer.next(data);
6                  observer.complete();
7              }, error =>{
8                  console.log(error);
9                  if (`${error}`.indexOf("Response with status: 0  for URL: null") >= 0){
10                     observer.error("Ops.. Servidor fora do ar!");
11                 } else {
12                     observer.error(error);
13                 }
14             });
15     })
16     .map((response) => response)
17     .timeoutWith(AppSettings.DEFAULT_TIMEOUT, Observable.throw(new Error('timeout!')))
18     .toPromise();
19 }
```

Fonte: elaborado pelo autor.

No Quadro 39, é apresentado o método `post`. Ele é responsável por executar chamadas HTTP do tipo POST para o servidor embarcado. Na linha 1 é informado na variável `postData` os dados enviados ao servidor. A variável `postData` é um JSON quando informada. Da mesma forma que no método `get`, o `post` também executa o método durante um determinado tempo, na linha 4 é feita essa configuração. Caso esse tempo seja excedido, é lançada uma exceção informando que o limite foi ultrapassado. Se a chamada for executada com sucesso, é retornado um objeto do tipo `Promise` contendo a resposta da requisição no objeto `Response`.

Quadro 39 - Método HTTP para chamadas do tipo POST

```
1  public post(url: string, postData, headersDict = null){
2      return this.http.post(`${url}`, postData, {headers: new Headers(headersDict)})
3          .map((response) => response)
4          .timeoutWith(AppSettings.DEFAULT_TIMEOUT, Observable.throw(new Error('timeout!')))
5          .toPromise();
6  }
```

Fonte: elaborado pelo autor.

3.2.5.2 Obter a Uniform Resource Locator (URL) do servidor embarcado

No Quadro 40 é apresentada a função responsável por obter a URL onde o servidor embarcado se encontra hospedado. Podemos observar na linha 4 que a aplicação busca as configurações. As configurações armazenam a URL na variável `endpoint` que é verificada na linha 6 e, caso não exista, é retornada a URL padrão da linha 1. Se existir a variável `endpoint` das configurações, a mesma é devolvida na função. O método retorna um objeto do tipo `Promise` que é utilizado pelo método que chamar esta função.

Quadro 40 - Obtenção da URL para o sistema embarcado

```
1  public static API_ENDPOINT_INIT = 'http://veiculo.sytes.net:5000/';
2  public getEndpoint(){
3      return new Observable<string>(observer => {
4          this.storage.get("configuracoes").then((result) => {
5              var _configuracoes = result ? result : {};
6              if(!_configuracoes.endpoint){
7                  _configuracoes.endpoint = AppSettings.API_ENDPOINT_INIT;
8                  observer.next(_configuracoes.endpoint);
9                  observer.complete();
10             } else
11                 observer.next(_configuracoes.endpoint);
12                 observer.complete();
13             }
14         }, error => {
15             observer.error(error);
16         })
17     })
18     .map((response) => response)
19     .toPromise();
20 }
```

Fonte: elaborado pelo autor.

3.2.5.3 Salvar e recuperar configurações

O Quadro 41 demonstra o código fonte utilizado para enviar as configurações ao servidor embarcado. Isto é feito na linha 2 através de uma requisição POST passando as configurações obtidas pelo aplicativo para o servidor. Na linha 4 e na linha 8 são armazenados os valores as configurações no banco de dados IndexedDB na chave `configuracoes`. Nota-se que, na linha 8, se o processo falhar ao salvar os dados no servidor, é armazenado mesmo assim no banco de dados para o seu envio posterior.

Quadro 41 - Salvando as configurações

```
1  this.appSettings.getEndpoint().then(endpoint => {
2    this.httpService.post(endpoint + "save_configs", this.configuracoes).then(result => {
3      if(result.json().status == "OK"){
4        this.storage.set("configuracoes", this.configuracoes);
5      }
6    }).catch(error =>{
7      this.storage.set("configuracoes", this.configuracoes);
8    });
9  });
```

Fonte: elaborado pelo autor.

No Quadro 42 é apresentado o método responsável por carregar as configurações armazenadas no servidor embarcado para a aplicação *mobile*. Isto é feito através de uma requisição GET ao servidor embarcado onde, é retornado essas configurações e salvas no dispositivo. Na linha 3 é demonstrado a obtenção das configurações e na linha 5 as mesmas são salvas no banco de dados IndexedDB na chave `configurações`.

Quadro 42 - Obtendo as configurações do servidor embarcado

```
1  this.appSettings.getEndpoint().then(endpoint => {
2    this.httpService.get(endpoint + 'get_configs').then(result => {
3      var configuracoes = result.json();
4      if(!configuracoes.error){
5        this.storage.set("configuracoes", this.configuracoes);
6      }
7    });
8  });
```

Fonte: elaborado pelo autor.

3.2.5.4 Mapas

O Quadro 43 demonstra o código responsável por chamar o servidor embarcado e buscar as coordenadas de localização. Na linha 2 é feita a chamada do tipo GET ao servidor no endpoint `/get_gps`, que retorna as coordenadas na linha 3. Na linha 8 e 9, essas coordenadas são salvas no banco de dados IndexedDB com a chave `localizacoes`. Já na

linha 10, as localizações são transferidas para a tela `LocalizacaoDetail` onde o mapa é apresentado.

Quadro 43 - Recuperação das coordenadas

```
1  this.appSettings.getEndpoint().then(endpoint => {
2    this.httpService.get(endpoint + 'get_gps').then(result => {
3      var coords = result.json();
4      var localizacao = {
5        data: this.datePipe.transform(new Date(), 'dd/MM/yyyy HH:mm:ss'),
6        coords: coords
7      };
8      this.localizacoes.push(localizacao);
9      this.storage.set("localizacoes", this.localizacoes);
10     this.navCtrl.push(LocalizacaoDetailPage, {localizacao: localizacao});
11   })
12 });
```

Fonte: elaborado pelo autor.

No Quadro 44 é apesentado o código fonte para exibir o mapa na aplicação *mobile*. Para isto é utilizada a biblioteca `angular-google-maps`, ela permite informar a latitude e longitude e ela desenha o mapa. Isto é feito na linha 2, onde, são passadas as coordenadas.

Quadro 44 - Código fonte de exibição do mapa

```
1  <agm-map [latitude]="localizacao.coords.lat" [longitude]="localizacao.coords.longit" [zoom]="17" >
2    <agm-marker [latitude]="localizacao.coords.lat" [longitude]="localizacao.coords.longit"></agm-marker>
3  </agm-map>
```

Fonte: elaborado pelo autor.

3.2.5.5 Sensores

Esta seção é responsável pela implementação na aplicação *mobile* dos sensores OBD2. No Quadro 45 é exposto o código referente à recuperação de PIDs. Na linha 3 é feita uma chamada HTTP ao servidor embarcado no *endpoint* `get_obdii_pids` que retorna uma lista de PIDs conforme Quadro 46. Na linha 5 esta lista é armazenada para a sua posterior utilização.

Quadro 45 - Obter PIDs

```
1  public recuperarPids(){
2    this.appSettings.getEndpoint().then(endpoint => {
3      this.httpService.get(endpoint + 'get_obdii_pids').then(result => {
4        var list = result.json();
5        this.pids = list;
6    })
7  });
```

Fonte: elaborado pelo autor.

No Quadro 46 é exposto um exemplo de JSON retornado pelo servidor embarcado quando são solicitados os PIDs. Nele consta informações como o código do PID, o nome do sensor e a unidade de medida. O valor será buscado posteriormente, por isso, este serviço disponibiliza todos os valores como 0.

Quadro 46 - Exemplo de PIDs em JSON


```
Array[7][
  {↔},
  {↔},
  {
    "codigo": "010D",
    "sensor": "Vehicle Speed",
    "unidade": "kph",
    "valor": "0"
  },
  {↔},
  {↔},
  {↔},
  {↔}
]
```


Fonte: elaborado pelo autor.

No Quadro 47 é apresentado o código para a obtenção de valores dos sensores retornados do servidor embarcado. Na linha 3 é realizada a chamada para o *endpoint* `get_obdii_values` que retorna, na linha 4, uma lista de valores JSON conforme Quadro 48. Observa-se que na linha 12 é executado novamente o método `monitorarPids`, ou seja, esta chamada é recursiva. Ela é utilizada para atualizar os valores sem precisar da interação do usuário. Na linha 6 até a linha 11, o objeto JSON é iterado e para cada `valor` retornado por ele é atualizado o `valor` com o mesmo código na variável `pids` que foi armazenado na chamada `get_obdii_pids`. Foi separada a chamada dos valores (Quadro 47) da chamada dos PIDs (Quadro 45) para otimizar a performance e economizar no plano de dados 3G.

Quadro 47 - Obter valores dos PIDs

```
1  public monitorarPids(){
2    this.appSettings.getEndpoint().then(endpoint => {
3      this.httpService.get(endpoint + 'get_obdii_values').then(result => {
4        var list = result.json();
5        this.valores = list;
6        this.valores.forEach(sensor => {
7          this.pids.filter(x => x.codigo == sensor.codigo)
8            .forEach(pid => {
9              pid.valor = sensor.valor;
10           });
11       });
12       this.monitorarPids();
13     })
14   });
15 }
```

Fonte: elaborado pelo autor.

No Quadro 48, é demonstrado um exemplo de JSON retornado à partir da chamada do *endpoint* `get_obdii_values` no servidor embarcado. Este contém uma lista de objetos que disponibiliza o código do PID e o seu respectivo valor lido da porta OBD2.

Quadro 48 - Retorno dos valores de PIDs em JSON

```
Array[7][
  {↔},
  {
    "codigo": "010C",
    "valor": "0"
  },
  {
    "codigo": "010D",
    "valor": "0"
  },
  {↔},
  {↔},
  {↔},
  {↔}
]
```

Fonte: elaborado pelo autor.

Por fim, no Quadro 49 é apresentado o código HTML do medidor de velocidade (velocímetro). A sua implementação na *tag* `radial-gauge` é feita através da biblioteca `GaugesModule`. Todos os atributos informados são para controle da própria framework para desenhar graficamente o medidor. Na linha 13 é informado o valor do sensor.

Quadro 49 - Código HTML do velocímetro

```
1  <radial-gauge *ngIf="sensor.unidade == 'kph'"
2      width="250" height="250" units="Km/h" min-value="0"
3      start-angle="90" ticks-angle="180" value-box="false"
4      major-ticks="0,20,40,60,80,100,120,140,160,180,200,220"
5      max-value="220" minor-ticks="2" stroke-ticks="true"
6      highlights='[
7        {"from": 160, "to": 220, "color": "rgba(200, 50, 50, .75)"}
8      ]'
9      color-plate="#fff" border-shadow-width="0" borders="false"
10     needle-type="arrow" needle-width="2" needle-circle-size="7"
11     needle-circle-outer="true" needle-circle-inner="false"
12     animation-duration="1500" animation-rule="linear"
13     [attr.value]="sensor.valor">
14 </radial-gauge>
```

Fonte: elaborado pelo autor.

3.2.5.6 Imagens

No Quadro 50 é apresentado o método para fazer a captura das imagens da câmera em tempo real. Isto é feito através de uma chamada no *endpoint* `get_video` do servidor embarcado. Na linha 3 é recuperado o endereço e armazenado na variável `src` que é utilizada pela *tag* HTML `img` do Quadro 51.

Quadro 50 - Método para *streamming* de imagens

```
1   public monitorarCamera(){
2     this.appSettings.getEndpoint().then(endpoint => {
3       this.src = endpoint + "get_video";
4       this.loader.dismiss();
5     });
6   }
```

Fonte: elaborado pelo autor.

Por fim, no Quadro 51 é demonstrada a linha responsável por desenhar as imagens recebidas pelo servidor embarcado em tempo real. Caso não receba nenhuma imagem, é atribuído o valor nulo ao atributo `src`.

Quadro 51 - HTML para renderizar as imagens em tempo real

```
1   <img [src]="src ? src : null" />
```

Fonte: elaborado pelo autor.

3.2.6 Operacionalidade da implementação

Nesta seção é exposta a operacionalidade da aplicação desenvolvida neste trabalho. É demonstrada a instalação do dispositivo no automóvel e, em seguida, apresentam-se as telas do aplicativo *mobile*.

A Figura 28 apresenta a instalação do dispositivo ao veículo, foi necessário conectar o adaptador ELM327 Bluetooth à tomada OBD2. O dispositivo foi acoplado ao retrovisor do automóvel para obter as imagens do interior do veículo. Por fim, o dispositivo foi conectado à tomada 12V do veículo onde o sistema embarcado inicia-se automaticamente.

Figura 28 – Instalação do dispositivo no veículo

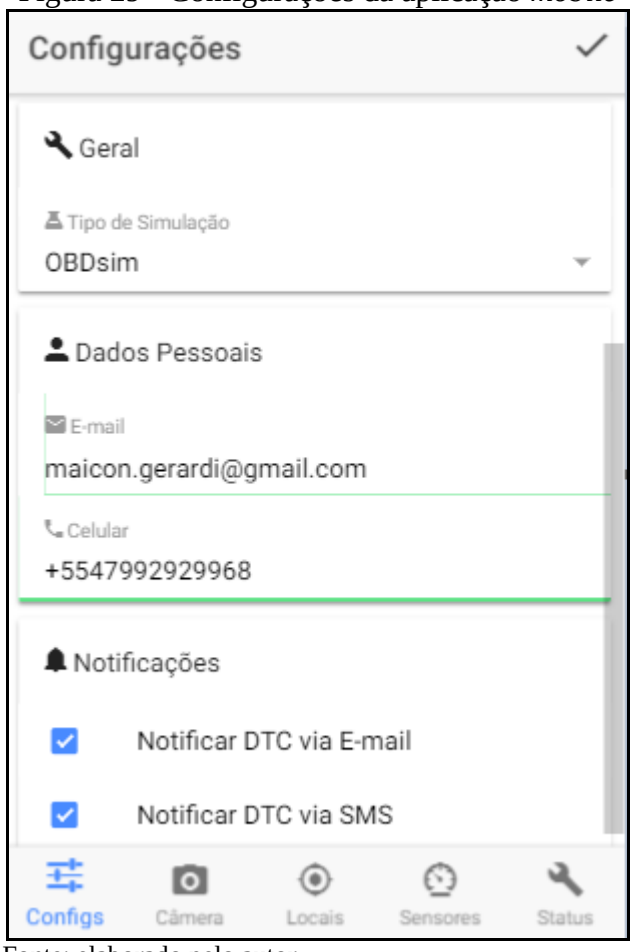
Fonte: elaborado pelo autor.

Depois da instalação concluída no automóvel, inicia-se a aplicação *mobile*. Na Figura 29, é ilustrada a tela de configurações que é a primeira tela do aplicativo. Nela é possível fazer os ajustes iniciais antes de utilizar a aplicação. São realizadas configurações de simuladores, dados pessoais do usuário e configurações de recebimento de notificações.

No campo de seleção `Tipo de Simulação` é possível informar se a realização da leitura da porta OBD2 é feita através de simulação. Ele dispõe das seguintes opções: `OBDsim`, `ECU Engine Pro` e `Sem simulador`. Quando é selecionado `Sem simulador`, a aplicação entende que está instalada em um veículo.

Na seção `Dados Pessoais`, o usuário pode informar o seu e-mail e celular. Já na seção `Notificações`, o usuário informa se deseja receber notificação de novos DTCs via e-mail e também via SMS. Após faze as configurações, deve-se clicar em `OK` (ícone do canto direito superior).

Figura 29 - Configurações da aplicação *mobile*

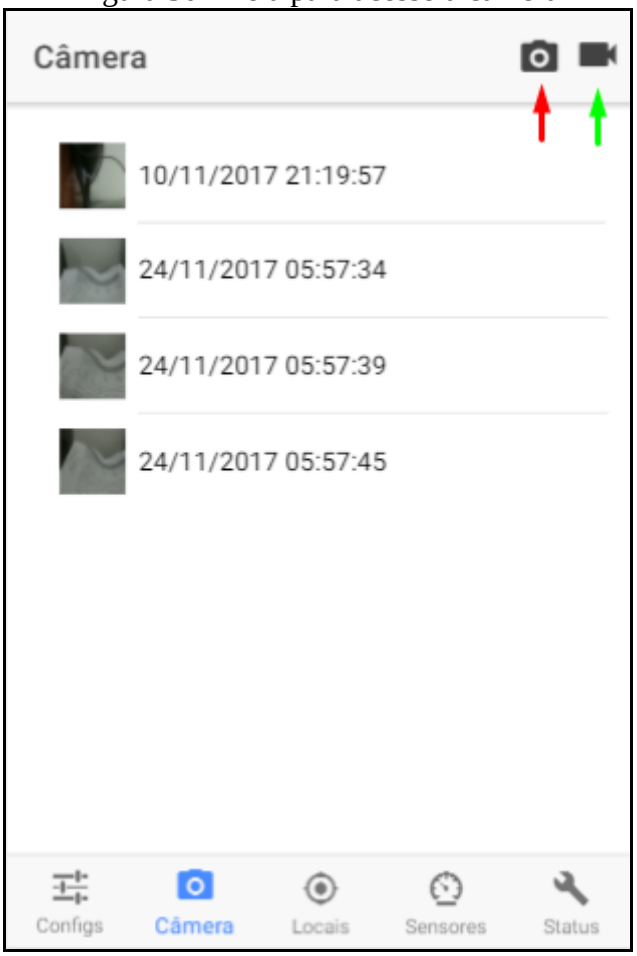


Fonte: elaborado pelo autor.

Na Figura 30, é apresentada a tela para acesso à câmera do dispositivo, nesta tela é apresentada uma lista com as fotos armazenadas no dispositivo, ao clicar em uma das fotos, o aplicativo direciona para a tela `Detalhes da Foto` conforme Figura 31 (a). É possível

capturar uma nova foto clicando no botão `Foto` (indicado pela seta vermelha) que, também é direcionada para a tela `Detalhes da Foto`. Por fim, é possível fazer capturas em tempo real da câmera, isto é realizado através do botão `Câmera` (indicado pela seta verde). Ao clicar no botão, o aplicativo redireciona para a tela de `Vídeo` (conforme Figura 31 (b)) com as imagens em tempo real.

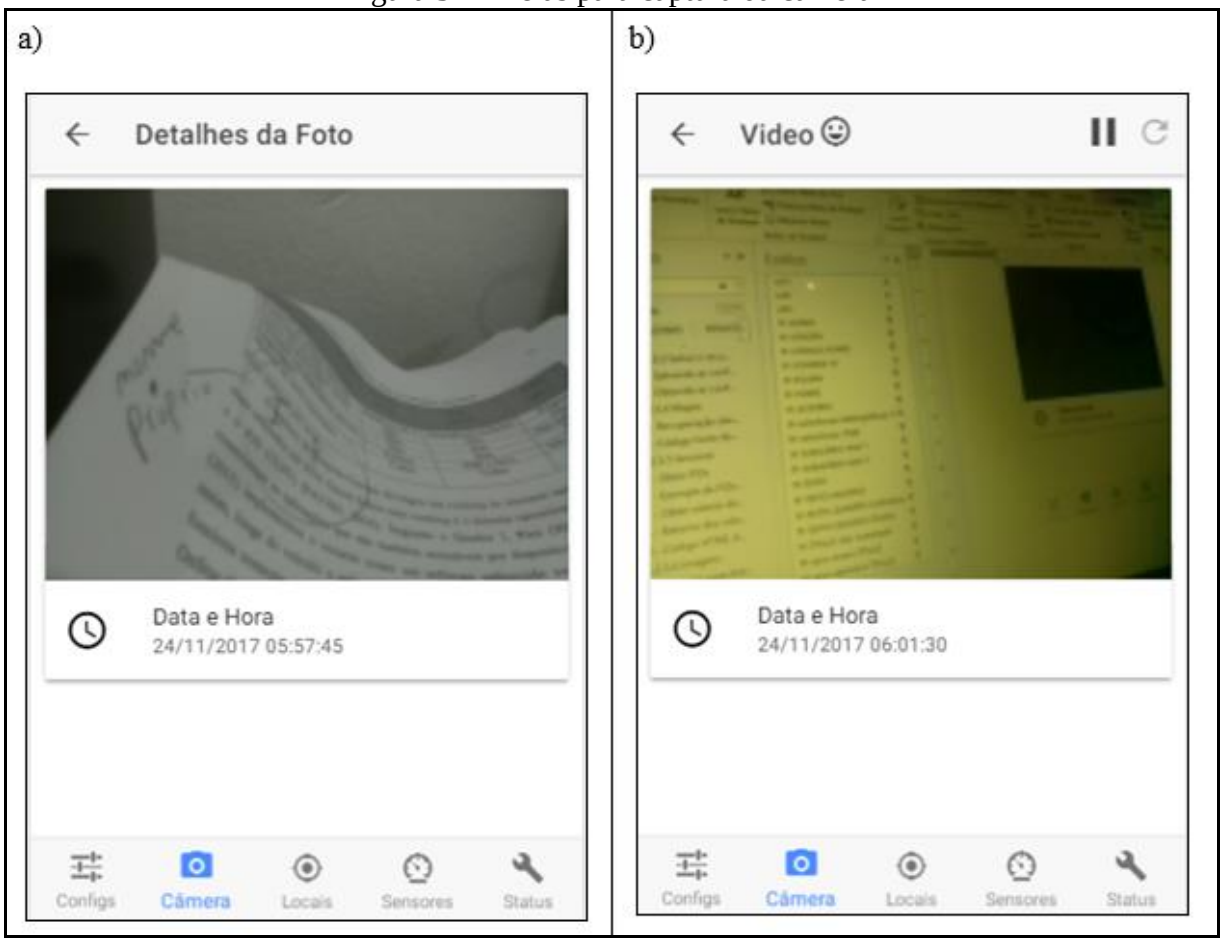
Figura 30 - Tela para acesso à câmera



Fonte: elaborado pelo autor.

Na Figura 31, são demonstradas duas telas, a tela do item (a) refere-se aos detalhes da foto selecionada ou nova foto extraída da aplicação. Já o item (b) ilustra as imagens capturadas em tempo real pela câmera. Ambas as telas apresentam data e a hora da capura.

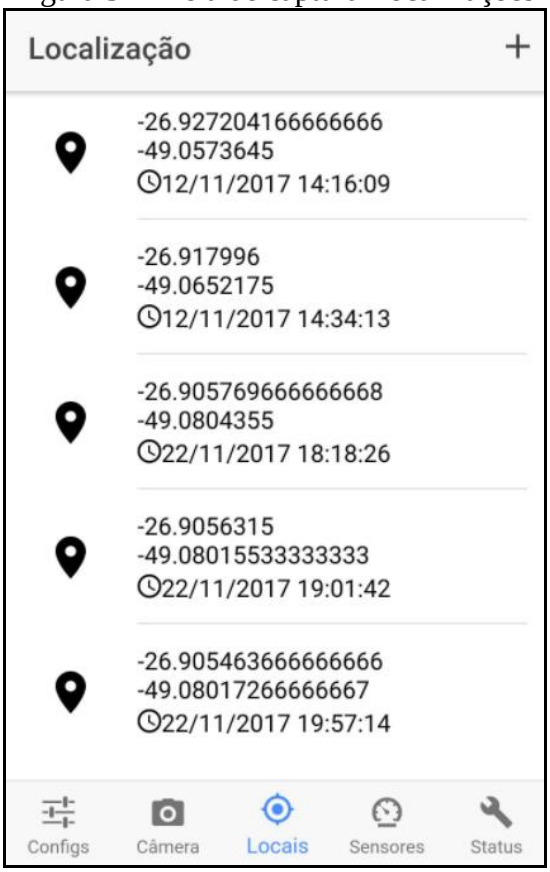
Figura 31 – Telas para captura da câmera



Fonte: elaborado pelo autor.

Na Figura 32 é apresentada a tela para a captura das localizações no dispositivo. Nela consta uma lista com as localizações já obtidas anteriormente, também há um botão "+" para fazer a captura de uma nova localização. Ao clicar em uma localização já capturada ou na obtenção de uma nova localização, o aplicativo redireciona para a tela de detalhamento da localização (Figura 33).

Figura 32 - Tela de capturar localizações



Fonte: elaborado pelo autor.

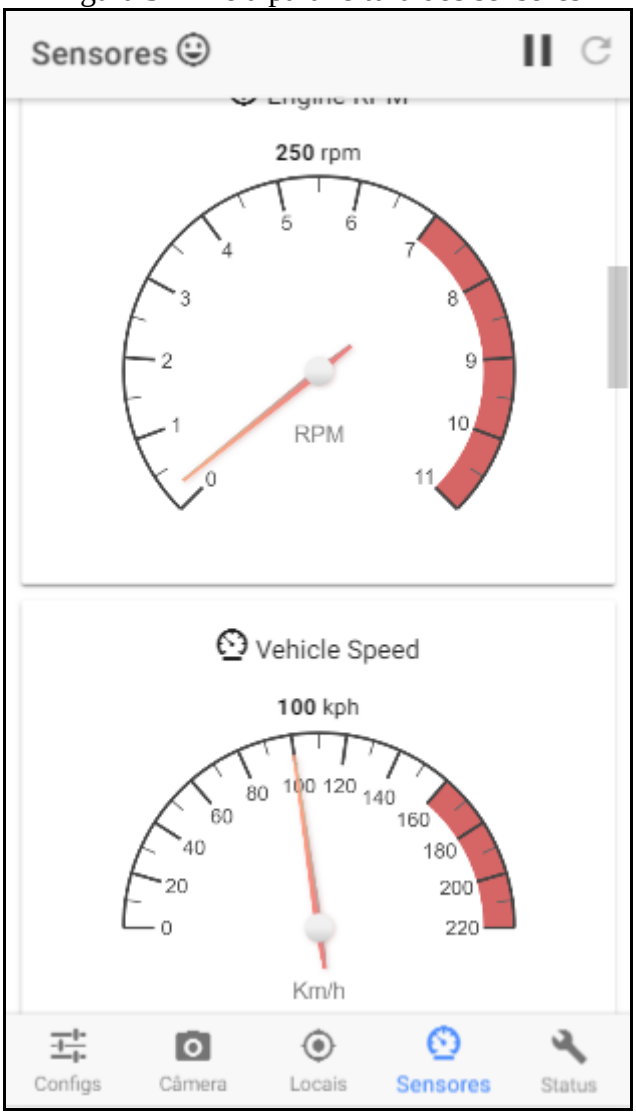
Na Figura 33 são detalhados os dados da localização selecionada ou extraída em tempo real. Nesta tela é apresentado o mapa indicando a devida localização, bem como dados como latitude, longitude e também, a data e hora da mesma.

Figura 33 - Detalhes da localização

Fonte: elaborado pelo autor.

Na Figura 34, é apresentado a tela para a leitura dos sensores do automóvel feitas em tempo real. São mostrados os medidores de temperatura, velocidade, rotação e os demais sensores encontram-se textualmente descritos. É possível pausar a leitura dos sensores caso necessário através do botão `Pause` (primeiro botão no canto direito superior). Também é possível restaurar a leitura dos sensores através do botão `Recarregar` (segundo botão no canto direito superior).

Figura 34 - Tela para leitura dos sensores



Fonte: elaborado pelo autor.

Por fim, na Figura 35 é ilustrada a tela responsável pela leitura dos códigos DTC do veículo. Esta tela contém três seções, sendo elas:

a) `Geral`: responsável por informa se a lâmpada MIL encontra-se ligada ou desligada e também, informar a quantidade de DTC.

b) `Códigos Pendentes`: informa os códigos DTC que estão pendentes de acender a MIL.

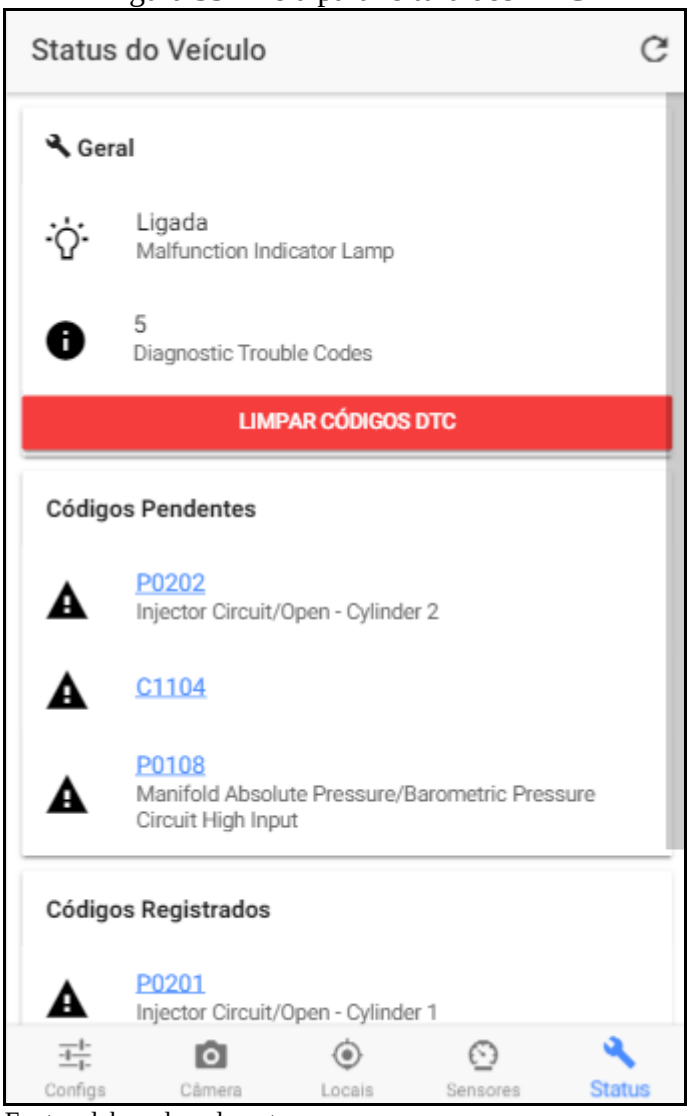
c) `Códigos Registrados`: códigos DTC que acenderam a MIL.

Para complementar a descrição destes códigos, ao clicar em um deles (qualquer código sublinhado em azul), é redirecionado o usuário à uma página[4] da Internet com a descrição e

[4] http://www.troublecodes.net/

causa do mesmo. Também é possível também fazer a limpeza destes códigos clicando no botão vermelho denominado `Limpar Códigos DTC`.

Figura 35 - Tela para leitura dos DTC



Fonte: elaborado pelo autor.

3.3 ANÁLISE DOS RESULTADOS

[Apresentar os casos de testes do software, destacando objetivo do teste, como foi realizada a coleta de dados e a apresentação dos resultados obtidos, preferencialmente em forma de gráficos ou tabelas, fazendo comentários sobre os mesmos.

Confrontar com os trabalhos correlatos apresentados na fundamentação teórica.]

4 CONCLUSÕES

[As conclusões devem refletir os principais resultados alcançados, realizando uma avaliação em relação aos objetivos previamente formulados. Deve-se deixar claro se os objetivos foram atendidos, se as ferramentas utilizadas foram adequadas e quais as principais contribuições do trabalho para o seu grupo de usuários ou para o desenvolvimento científico/tecnológico.]

[Deve-se também incluir aqui as principais vantagens do seu trabalho e limitações.]

4.1 EXTENSÕES

[Sugestões para trabalhos futuros.]

REFERÊNCIAS

ALIEXPRESS. **Super Mini Elm327 Bluetooth OBD2 V2.1 Elm 327 V 2.1**. 2017. Disponível em: <https://pt.aliexpress.com/store/product/14000mAh-Mini-portable-car-jump-starter-multi-function-diesel-power-bank-bateria-battery-12V-car-charger/737379_32757909248.html>. Acesso em: 17 nov. 2017.

ALMEIDA, Eduardo Luciano de; FARIA, Felipe Freitas de. **Scanner OBD-II Em Plataforma LabView**. 2013. 139 f. TCC (Graduação) - Curso de Tecnologia em Eletrônica Automotiva, Faculdade de Tecnologia Fatec Santo André, São Paulo, 2013.

Aplicações De Automação Em Iot - Internet Of Things. Extrema - Minas Gerais: Editorial E-locução, v. 10, 2016. Anual. Disponível em: <http://periodicos.faex.edu.br/index.php/e-locucao/article/view/104>. Acesso em: 27 out. 2017.

BARROS, Eduardo. **Estudo das diferenças dos requerimentos das principais legislações on board diagnostics para padronização de testes de desenvolvimento e validação de transmissão automática de automóveis**. 2012. 59 f. Monografia (Especialização) - Curso de Engenharia Automotiva, Centro Universitário, Centro Universitário do Instituto Mauá de Tecnologia, São Caetano, 2012.

BAUMGARTEN, Nykolas E. A., **FINDCAR**: RASTREADOR VEICULAR UTILIZANDO OPENWRT. 2016, 56 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

CCM. **Codificação Base64**. 2017. Disponível em: <http://br.ccm.net/contents/55-codificacao-base64>. Acesso em: 21 nov. 2017.

CZERWONKA, Mariana, **Falta de manutenção triplica risco de acidentes**. [Goiás?], 2016. Disponível em <http://portaldotransito.com.br/noticias/falta-de-manutencao-triplica-risco-de-acidentes>. Acesso em: 20 mar 2017.

DEAL EXTREME. **GY-NEO6MV2 Flight Controller GPS Module - Blue**. 2017. Disponível em: <http://www.dx.com/p/gy-neo6mv2-flight-controller-gps-module-blue-232595#.Wg7RakqnHcc>. Acesso em: 17 nov. 2017.

DIAS, Renata. R. F. **Internet das coisas sem mistérios**: uma nova inteligência para os negócios. São Paulo; Netpress Books, 2016

ELM ELECTRONICS. **OBD**. 2017. Disponível em: <https://www.elmelectronics.com/products/ics/obd/?v=19d3326f3137>. Acesso em: 31 out. 2017.

G1 Ribeirão e Franca, **Estatística divulgada pela SSP mostra aumento da violência em Ribeirão**. [São Paulo], 2017. Disponível em: <http://g1.globo.com/sp/ribeirao-preto-franca/noticia/2017/02/estatistica-divulgada-pela-ssp-mostra-aumento-da-violencia-em-ribeirao.html>. Acesso em: 20 mar 2017.

GIZMODO. **What's this 1080p Stuff?** 2013. Disponível em: <https://gizmodo.com/160103/tuning-fork>. Acesso em: 17 nov. 2017.

GRINBERG, Miguel. **Video Streaming with Flask**. 2014. Disponível em: <https://blog.miguelgrinberg.com/post/video-streaming-with-flask>. Acesso em: 21 nov. 2017.

MACHADO, António S. L.; OLIVEIRA, Bruno R. R., **O Sistema OBD (On-Board Diagnosis)**. 2007, 8 f, Mestrado em Automação e Sistemas, Instituto Superior de Engenharia do Porto, Porto - Portugal.

MANCINI, Mônica. **Internet das Coisas: História, Conceitos**, Aplicações e Desafios. 2017. Disponível em: <https://pmisp.org.br/documents/acervo-arquivos/241-internet-das-coisas-historia-conceitos-aplicacoes-e-desafios/file>. Acesso em: 17 nov. 2017.

MCCORD, Kleint. **Automotive Diagnostic System**: Understanding OBD-I & OBD-II. North Branch: CarTech, 2011.

MULTILÓGICA SHOP. **Módulo GPS GY-NEO6MV2 com conversor de sinal**. 2017. Disponível em: <https://multilogica-shop.com/modulo-gps-gy-neo6mv2-com-conversor-de-sinal>. Acesso em: 17 nov. 2017.

NO-IP. **How to Install the No-IP DUC on Raspberry Pi**: How to Install the No-IP DUC on Raspberry Pi. 2017. Disponível em: <http://www.noip.com/support/knowledgebase/install-ip-duc-onto-raspberry-pi/>. Acesso em: 21 nov. 2017.

NULL BYTE. **Set Up Kali Linux on the New $10 Raspberry Pi Zero W**. 2017. Disponível em: <https://null-byte.wonderhowto.com/how-to/set-up-kali-linux-new-10-raspberry-pi-zero-w-0176819/>. Acesso em: 25 maio 2017.

OBD SOLUTIONS. **What is obd?**. [2017?]. Disponível em: <http://www.obdsol.com/knowledgebase/on-board-diagnostics/what-is-obd/>. Acesso em: 24 maio 2017.

OBDSIM. **OBDSim**. 2017. Disponível em: <https://icculus.org/obdgpslogger/obdsim.html>. Acesso em: 21 nov. 2017.

ODEGA, Alessandra, **Confira os modelos de veículos mais roubados e quanto custa o seguro de cada um deles**. Santa Catarina, 2016. Disponível em: <http://ndonline.com.br/florianopolis/coluna/alessandra-ogeda/confira-os-modelos-de-veiculos-mais-roubados-e-quanto-custa-o-seguro-de-cada-um-deles>. Acesso em: 20 mar 2017.

OUTILS OBD FACILE. **ELM327 Car Diagnostics Interfaces**. 2017. Disponível em: <https://www.outilsobdfacile.com/diagnostic-interface-elm-327.php>. Acesso em: 12 nov. 2017.

PACHECO, Lucas V., **Monitoramento e gestão de uma frota de veículos utilizando sistemas embarcados**. 2016, 76 f. Trabalho de Conclusão de Curso (Curso de Engenharia Elétrica) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Paulo.

PI SUPPLY. **Raspberry Pi Camera Board v1.3 (5MP, 1080p)**. 2017. Disponível em: <https://www.pi-supply.com/product/raspberry-pi-camera-board-v1-3-5mp-1080p/>. Acesso em: 17 nov. 2017.

PINA, Afonso L. P., **Sistema De Localização De Veículos Para Smartphone Android**. 2015, 136 f, Tese/Dissertação de Mestrado (Engenharia Eletrotécnica e de Computadores) – Departamento de Engenharia Eletrotécnica, Instituto Superior de Engenharia do Porto, Porto - Portugal.

PYTHON-OBD. **Python-OBD**. 2017. Disponível em: <http://python-obd.readthedocs.io/en/latest/>. Acesso em: 22 nov. 2017.

QUOC, Chinh Luong. **ECU Engine Pro**. 2017. Disponível em: <https://play.google.com/store/apps/details?id=com.obdii_lqc.android.obdii.elm327.ecusimpro>. Acesso em: 21 nov. 2017.

RASPBERRY PI FOUNDATION, **FAQS**. [2017?]. Disponível em: <https://www.raspberrypi.org/help/faqs/>. Acesso em: 01 abr. 2017.

SANTOS, Arthur Luis V., **Economia de combustível com o uso de telemetria para veículos de passeio**. 2016, 82 f. Trabalho de Conclusão de Curso (Engenharia de Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Rio Grande do Sul.

SANTOS, Bruno P. et al. **Internet das Coisas**: da Teoria à Prática, [2016?], 50 f. Artigo científico. Departamento de Ciência da computação, Universidade Federal de Minas Gerais

SOUZA, Beatriz, **Quatro em cada dez veículos de carga apresentam falha mecânica**. [Santa Catarina?], 2016. Disponível em: <http://www.perkons.com.br/noticia/1694/quatro-em-cada-dez-veiculos-de-carga-apresentam-falha-mecanica>. Acesso em: 20 mar 2017.

STAROSKY, Ricardo A., **OBD-JRP**: monitoramento veicular com java e raspberry pi. 2016, 87 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

THOMSEN, Adilson. **Novo Raspberry Pi Zero W com Wifi e Bluetooth**. 2017. Disponível em: <https://www.filipeflop.com/blog/raspberry-pi-zero-w-com-wifi-e-bluetooth/>. Acesso em: 17 nov. 2017.

TOP CAR DIAGNOSTICS SUPPORT. **Description of OBD Communication Standards / Protocols**. 2014. Disponível em: <http://www.totalcardiagnostics.com/support/Knowledgebase/Article/View/74/0/description-of-obd-communication-standards--protocols>. Acesso em: 31 out. 2017.

ZTE BRASIL. **MF626**. 2012. Disponível em: <http://www.ztebrasil.com.br/products/mobile/detail/Portugal/Portugal_mobile/detail/201009/t20100930_232495.html>. Acesso em: 17 nov. 2017.

APÊNDICE A – Relação dos componentes utilizados

A Tabela 1 contém todos os componentes utilizados para elaboração do projeto, o seu valor unitário e o custo total. Os componentes com valor em dólar foram comprados nos sites Ali Express e taxa de câmbio foi de R$3,30 para 1 dólar. Os componentes com valor em real foram comprados no site da Proesi e no Mercado Livre. A taxa de frete do produto já está inclusa no valor total.

Tabela 1 - Relação dos componentes utilizados

Peça	Quantidade	Valor Unitário	Valor Total (R$)
Raspberry Pi Zero W	1	U$ 26	85,80
WAVGAT Raspberry Pi Camera	1	U$ 6	19,80
Adaptador ELM327 Bluetooth V1.5	1	U$ 6	19,80
Módulo GPS GY NEO6MV2-GPS6MV2	1	U$ 10	33,00
Cabo Flex Câmera Raspberry Pi Zero	1	U$ 6	19,80
Modem USB ZTE MF626	1	R$ 9	9,00
Cabo conector de jumper fêmea x fêmea	14	R$ 0,35	4,90
Barramento para jumpers	1	R$ 1	1,00
Resistor 220 ohms	5	R$ 0,10	0,50
LED alto brilho	5	R$ 1,30	6,50
Placa universal de fenolite	1	R$ 10	10,00
Caixa patola 7x5	1	R$ 12	12,00
Chip pré-pago operadora TIM	1	R$ 10	10,00
Botão switch	2	R$ 1	2,00
Cartão micro SD 16 GB	1	R$ 25	25,00
Adaptador Micro USB para USB	1	R$ 22	22,00
		TOTAL	**R$ 281,10**

Fonte: elaborado pelo autor.

APÊNDICE B – Relação das bibliotecas externas utilizadas no sistema embarcado

Neste apêndice é apresentada a relação de bibliotecas externas utilizadas pelo sistema embarcado. Foram necessárias bibliotecas baixadas pelo gerenciados de pacotes Python denominado PIP, bibliotecas baixadas pelo gerenciador de aplicações do sistema operacional Raspian chamado apt-get e, por fim, bibliotecas baixadas do site GitHub.

Abaixo são listadas as bibliotecas baixadas pelo gerenciador de pacotes PIP:

a) twilio;
b) flask;
c) smtplib;
d) obd.

Foram utilizadas bibliotecas baixadas pelo gerenciador de aplicações apt-get do sistema operacional Raspian. Essas bibliotecas são baixadas através dos comandos apresentados no Quadro 52.

Quadro 52 - Comandos executados

```
sudo apt-get install python-rpi.gpio python3-rpi.gpio
sudo apt-get install bluetooth bluez-utils blueman
sudo apt-get install ppp usb-modeswitch wvdial
```

Fonte: elaborado pelo autor.

Para a instalação do serviço de DDNS No-Ip, foram executado os comandos do Quadro 53.

Quadro 53 - Comandos para a instalação do No-IP

```
mkdir /home/pi/noip
cd /home/pi/noip
wget http://www.no-ip.com/client/linux/noip-duc-linux.tar.gz
tar vzxf noip-duc-linux.tar.gz
cd noip-2.1.9-1
sudo make
sudo make install
sudo /usr/local/bin/noip2
sudo noip2 -S
```

Fonte: No-IP (2017, p. 1)

Além disso, foram necessárias outras duas bibliotecas externas baixadas do site GitHub que são:

a) python-obd: responsável pela leitura dos dados da porta OBD2;
b) flask-video-streaming: disponibilizar dados da câmera do Raspberry Pi via streamming.