

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO
CÂMPUS CAMPINAS

RUAN LUIZ ALVES DA SILVA

**APLICAÇÃO DA IOT EM SISTEMAS EMBARCADOS AUTOMOTIVOS PARA
MONITORAMENTO E DETECÇÃO DE FALHAS**

CAMPINAS

2017

RUAN LUIZ ALVES DA SILVA

**APLICAÇÃO DA IOT EM SISTEMAS EMBARCADOS AUTOMOTIVOS PARA
MONITORAMENTO E DETECÇÃO DE FALHAS**

Trabalho de Conclusão de Curso apresentado como exigência parcial para obtenção do diploma do Curso de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia Câmpus Campinas.

Orientador: Ricardo Barz Sovat

CAMPINAS

2017

Solicitar a ficha catalográfica pelo sistema Pergamum, (Meu Pergamum, solicitações, ficha catalográfica) após as correções sugeridas pela banca.

Apresenta-se no verso da página de rosto

da Silva, Ruan Luiz Alves.

**APLICAÇÃO DA IOT EM SISTEMAS EMBARCADOS AUTOMOTIVOS
PARA MONITORAMENTO E DETECÇÃO DE FALHAS / RUAN LUIZ ALVES
DA SILVA. – 2017.**

63 f. : il.

Orientador: Ricardo Barz Sovat

Trabalho Final de Curso – INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO, CÂMPUS CAMPINAS. , 2017.

1. Palavra-chave. 2. Palavra-chave. 3. Palavra-chave. I. Sobrenome, Nome do orientador, orient. II. Título.

RUAN LUIZ ALVES DA SILVA

**APLICAÇÃO DA IOT EM SISTEMAS EMBARCADOS AUTOMOTIVOS PARA
MONITORAMENTO E DETECÇÃO DE FALHAS**

Trabalho de Conclusão de Curso apresentado
como exigência parcial para obtenção do di-
ploma do Curso de Tecnologia em Análise
e Desenvolvimento de Sistemas do Instituto
Federal de Educação, Ciência e Tecnologia
Câmpus Campinas.

Aprovado pela banca examinadora em: 04 de dezembro de 2017

BANCA EXAMINADORA

Prof. Dr. Ricardo Barz Sovat (orientador)
IFSP Câmpus Campinas

Prof. Me. André Willik Valenti
IFSP Câmpus Campinas

Prof. Dr. Andreiwid Sheffer Correa
IFSP Câmpus Campinas

RESUMO

Este trabalho tem o objetivo de realizar um estudo a respeito das tecnologias embarcadas presentes nos automóveis para entender os padrões de comunicação entre os sistemas adotados por eles, e como a informática pode contribuir para tornar as informações mais inteligíveis e facilitar ou automatizar algumas análises. Propõe-se a implementação de um sistema capaz de ler as informações provenientes nos sensores dos automóveis e comunicar-se com serviços de computação em nuvem. O desenvolvimento desse sistema é dividido em três frentes distintas: a primeira frente é relacionada ao desenvolvimento do software responsável por ler as informações do automóvel; a segunda frente é responsável por preparar o ambiente do *Raspberry Pi* para execução, incluindo a parte de configuração do dispositivo e a implantação do software, e a terceira frente está relacionada com a integração do software com a computação em nuvem. O desenvolvimento do software utilizou a linguagem Java, programação orientado a objetos, além do padrão *Model-View-Controller (MVC)*. No serviço de computação em nuvem foram utilizados o banco de dados MongoDB e o framework Node.js para a criação do web service. O sistema permite gerar dados que serão armazenados em nuvem e poderão ser disponibilizados para outras aplicações envolvendo Internet das Coisas e *Big Data*. O resultado final foi obtido com o sistema gerando dados do veículo em funcionamento, permitindo o monitoramento e possíveis análises automatizadas via web.

Palavras-chave: Sistemas embarcados automotivos. Monitoramento veicular. Internet das coisas. *Big data*.

ABSTRACT

This project aims at studying embedded technologies found in cars in order to understand the communication patterns among the systems adopted by the automotive industry. Also, intends to explore means by which computer science can contribute to make clear the information involved and simplify or automate some analyses. In order to apply the concepts found, it was proposed the implementation of a system upon a low cost hardware platform (using a Raspberry Pi board) able to read information originated from cars' sensors and to communicate with cloud computing services. This systems development was divided in three distinct fronts. The first one is related to the software in charge of obtain the vehicle's data; the second is responsible for prepare the Raspberry Pi's environment to execute, including the device configuration and software installation. The last one deals with the integration between de embedded software and the cloud computing platform. The whole development was designed under the object oriented paradigm, using Java, as well as architectural patterns Model-View-Controller (MVC) and Data Access Object (DAO). The cloud computing services adopted have used MongoDB as database management system and the Node.js framework for web service creation. The system allows generating data to be stored in the cloud and which can be made available to other applications involving diverse areas as, for example, Internet of Things and Big Data. The final prototype succeeds in getting data from a vehicle in operation, permitting its tracking and potential automated analyses over the WWW.

Keywords: Automotive embedded systems. Vehicular tracking. Internet of things. Big Data.

LISTA DE FIGURAS

Figura 1 – Representação das cinco categorias funcionais.	21
Figura 2 – Representação do monitoramento de RPM e a comunicação entre as categorias.	23
Figura 3 – Representação da atuação dos sensores e atuadores.	23
Figura 4 – Foto de uma <i>ECU</i> .	24
Figura 5 – Diagrama da comunicação entre <i>ECU</i> , sensor e atuador.	25
Figura 6 – Representação de redes distintas interligadas.	26
Figura 7 – Representação de outros protocolos implementados no padrão <i>CAN</i> .	28
Figura 8 – Variações do protocolo SAE J1850.	28
Figura 9 – Foto do conector <i>OBD-II</i> .	29
Figura 10 – Foto da luz <i>MIL</i> do painel.	30
Figura 11 – Diagrama representando a arquitetura da rede veicular.	30
Figura 12 – Foto do ELM327 <i>Bluetooth</i> .	31
Figura 13 – Diagrama representando a interação do ELM327 com a rede automotiva.	32
Figura 14 – Foto do <i>Raspberry Pi 3</i> .	34
Figura 15 – Representação da interação das 'coisas' dentro do conceito de <i>IoT</i> .	36
Figura 16 – Imagem da tela de leitura prototipada.	37
Figura 17 – Diagrama de pacote exibindo a estrutura de pacotes do projeto com as respectivas classes.	38
Figura 18 – Foto da classe ELM327.	39
Figura 19 – Foto do método <i>disconnect</i> da classe ELM327.	39
Figura 20 – Foto do método <i>readRpm</i> da classe ELM327.	39
Figura 21 – Foto do método <i>readSpeed</i> da classe ELM327.	40
Figura 22 – Foto do método <i>readFuelPressure</i> da classe ELM327.	40
Figura 23 – Foto do método <i>readOilTemp</i> da classe ELM327.	40
Figura 24 – Foto do método <i>readFindFuelType</i> da classe ELM327.	40
Figura 25 – Foto do método <i>readFuelLevel</i> da classe ELM327.	40
Figura 26 – Foto do método <i>clearBuffer</i> da classe ELM327.	41
Figura 27 – Representação da arquitetura <i>MVC</i> do projeto.	41
Figura 28 – Foto do display do <i>Raspberry Pi</i> .	42
Figura 29 – Porta GPIO do <i>Raspberry Pi</i> .	42
Figura 30 – Software sendo executado no <i>Raspberry Pi</i> .	43
Figura 31 – Foto da estrutura do banco de dados.	44
Figura 32 – Diagrama de pacote exibindo a nova estrutura de pacotes do projeto com as respectivas classes.	45
Figura 33 – Representação da arquitetura <i>MVC</i> com o padrão <i>DAO</i> do projeto.	45
Figura 34 – Foto da classe <i>ELM327ReadSensors</i> .	46

Figura 35 – Foto da classe <i>DBConnection</i>	46
Figura 36 – Foto da classe <i>ELM327ReadSensorsDAO</i>	47
Figura 37 – Foto do método js responsável por responder a solicitações <i>GET</i> na rota ‘/collection’.	48
Figura 38 – Foto dos elementos de filtro da página.	48
Figura 39 – Foto do método <i>JQuery</i> responsável pelo <i>AJAX</i>	49
Figura 40 – Diagrama representando a comunicação do sistema com a nuvem.	49
Figura 41 – Foto dos objetos <i>JSON</i> disponibilizados pelo <i>web service</i>	52
Figura 42 – Foto da página web estruturando em uma tabela os dados <i>JSON</i>	53
Figura 43 – Foto do método <i>getConnectionBluetooth</i> da classe <i>BluetoothConnection</i> . . .	59
Figura 44 – Foto da Classe <i>DiscoveryDevices</i> implementando a interface <i>DiscoveryListener</i> . .	60
Figura 45 – Foto do método <i>deviceDiscovered</i> da classe <i>DiscoveryDevices</i>	60
Figura 46 – Foto do método <i>inquiryCompleted</i> da classe <i>DiscoveryDevices</i>	61
Figura 47 – Foto do método <i>servicesDiscovered</i> da classe <i>DiscoveryDevices</i>	61
Figura 48 – Foto do método <i>serviceSearchCompleted</i> da classe <i>DiscoveryDevices</i>	61
Figura 49 – Foto do método <i>discovery</i> da classe <i>DiscoveryDevices</i>	62
Figura 50 – Foto da classe <i>ConnectToDevice</i>	62
Figura 51 – Foto do método <i>connectToDevice</i> da classe <i>ConnectToDevice</i>	63

LISTA DE TABELAS

Tabela 1 – Tipos de redes e suas respectivas velocidades de transmissão de acordo com a <i>SAE</i> .	26
---	----

LISTA DE SIGLAS

API	<i>Application Programming Interface</i>
AWS	<i>Amazon Web Service</i>
CAN	<i>Controller Area Network</i>
CASAGRAS	<i>Coordination And Support Action for Global RFID-related Activities and Standardisation</i>
CPS	<i>Cyber-Physical Systems</i>
DAO	<i>Data Access Object</i>
DLC	<i>Diagnostic Link Connector</i>
DTC	<i>Diagnostic Trouble Codes</i>
EBS	<i>Elastic Block Store</i>
EC2	<i>Elastic Compute Cloud</i>
ECU	<i>Electronic Control Unit</i>
HMI	<i>Human-Machine Interface</i>
IaaS	<i>Infrastructure as a Service</i>
IOT	<i>Internet Of Things</i>
JDK	<i>Java Development Kit</i>
JRE	<i>Java Runtime Environment</i>
LIN	<i>Local Interconnect Protocol</i>
MIL	<i>malfunction indicator lamp</i>
MOST	<i>Media Oriented Systems Transport</i>
MVC	<i>Model View Controller</i>
OBD	<i>OnBoard Diagnostics</i>
PaaS	<i>Platform as a Service</i>
PCM	<i>powertrain control module</i>

PID	<i>parameter identification</i>
PWM	<i>pulse width modulation</i>
SaaS	<i>Software as a Service</i>
SAE	<i>Society for Automotive Engineers</i>
SBC	<i>Single-Board Computer</i>
SOC	<i>System on a Chip</i>
TCM	<i>Transmission Control Module</i>
TCU	<i>Transmission Control Unit</i>
VAN	<i>Vehicle Area Network</i>
VPW	<i>variable pulse width</i>

SUMÁRIO

SUMÁRIO	11	
1	INTRODUÇÃO	13
1.1	CONTEXTUALIZAÇÃO	13
1.2	MOTIVAÇÃO	14
1.3	DEFINIÇÃO DO PROBLEMA	14
1.4	ABORDAGEM PROPOSTA	14
1.5	TRABALHOS RELACIONADOS	15
2	JUSTIFICATIVA	16
3	OBJETIVOS	17
3.1	OBJETIVO GERAL	17
3.2	OBJETIVOS ESPECÍFICOS	17
4	FUNDAMENTAÇÃO TEÓRICA	19
4.1	SISTEMAS E SOFTWARE EMBARCADOS	19
4.2	SISTEMAS DISTRIBUÍDOS	19
4.3	ARQUITETURA DO SISTEMA AUTOMOTIVO	20
4.3.1	CATEGORIAS FUNCIONAIS DOS SISTEMAS EMBARCADOS AUTOMOTIVOS	20
4.3.2	<i>ECU (ELECTRONIC CONTROL UNIT)</i>	24
4.3.3	PROTOCOLOS AUTOMOTIVOS	25
4.3.3.1	PROTÓCOLO CAN (<i>CONTROLLER AREA NETWORK</i>)	27
4.3.3.2	PROTÓCOLO SAE J1850	28
4.3.4	CONECTOR <i>OBD-II (ONBOARD DIAGNOSTICS)</i>	29
4.4	ELM327	30
4.4.1	COMANDOS AT	32
4.4.2	COMANDOS <i>OBD</i>	32
4.4.2.1	COMUNICAÇÃO COM O VEÍCULO - PADRÕES DE SOLICITAÇÃO	33
4.5	<i>RASPBERRY PI</i>	33
4.6	COMPUTAÇÃO EM NUVEM	34
4.7	INTERNET DAS COISAS (<i>INTERNET OF THINGS - IOT</i>)	35
5	METODOLOGIA	37
5.1	DESENVOLVIMENTO DO SOFTWARE DE LEITURA	37

5.1.1	ARQUITETURA DO SOFTWARE	38
5.2	CONFIGURAÇÃO DO <i>RASPBERRY PI</i> E INSTALAÇÃO DO SOFTWARE	41
5.3	INTEGRAÇÃO DA APLICAÇÃO COM SERVIÇO DE COMPUTAÇÃO EM NUVEM	43
5.3.1	INSTALAÇÃO E CONFIGURAÇÃO DA BASE DE DADOS NA NUVEM	44
5.3.2	INTEGRAÇÃO DO SOFTWARE COM A BASE DE DADOS	44
5.3.3	INSTALAÇÃO E CRIAÇÃO DO <i>WEB SERVICE</i>	47
5.3.4	INSTALAÇÃO E CONFIGURAÇÃO DO SERVIDOR WEB E CRIAÇÃO DA PÁGINA	48
6	TESTES E AVALIAÇÃO DOS RESULTADOS	50
6.1	TESTE DO SOFTWARE DE LEITURA	50
6.2	TESTE DE INTEGRAÇÃO DO SOFTWARE COM O <i>RASPBERRY PI</i> . .	51
6.3	TESTES DE INTEGRAÇÃO DA APLICAÇÃO COM A COMPUTAÇÃO EM NUVEM	51
7	CONCLUSÕES	54
7.1	PROPOSTA PARA TRABALHOS FUTUROS	54
A	REFERÊNCIAS	56
A	Apêndice A	59

1 INTRODUÇÃO

Nesta seção será abordado a contextualização e a motivação que levou a realização e desenvolvimento deste trabalho.

1.1 CONTEXTUALIZAÇÃO

Existem na atualidade diversos tipos e modelos de automóveis possuindo diversas tecnologias integradas a eles. Entretanto, nem sempre foi assim. No início da história automotiva, surgem os primeiros carros a manivela em meados de 1880 e logo após chegam os carros a combustão interna. Depois de um tempo surgiram os carros carburados e depois de um certo período, chegaram os carros com injeção eletrônica. É observado que a cada período que se passa, o automóvel ganha alguns itens e tecnologias novas com a finalidade de melhorar o desempenho, combinando baixo consumo e baixa emissão de poluentes. Independente do ano ou modelo do veículo, ou até mesmo a tecnologia adotada por ele, o desgaste natural das peças é comum a qualquer modelo de automóvel. Vale ressaltar que estes desgastes podem ser agravados dependendo da forma com que o motorista utiliza seu veículo. Independentemente da causa do desgaste, quando há algum problema, geralmente o automóvel passa a apresentar um comportamento fora do comum, o que indica uma possível falha. Quando esses comportamentos são emitidos na forma de ruídos, há uma certa facilidade na percepção de que algo está errado, e uma possível manutenção ou revisão deve ser feita. Porém, quando esses comportamentos não fazem emissão de nenhum sinal ou ruído aparente, existe uma certa dificuldade na percepção de algum eventual defeito.

Os veículos que circulam atualmente possuem diversos sensores e controladores que fazem parte de um sistema embarcado automotivo. Esses sensores são responsáveis basicamente por coletar algumas informações do veículo e automatizar algum funcionamento específico do automóvel. Um exemplo típico é o controle de injeção de combustível, que é feito de forma eletrônica na maioria dos carros que trafegam pelas cidades e estradas. Ele evita o consumo excessivo de combustível durante um trajeto percorrido. Outra situação comum se encontra no painel de instruções, onde são mostradas algumas informações limitadas que são monitorados pelos sensores e exibidas ao condutor, como a rotação do motor, indicador de velocidade, temperatura do motor, entre outros.

Analizando os veículos atuais, nota-se que parte de seu funcionamento está deixando de ser apenas mecânico, e passando a ser controlado por sistemas eletrônicos. Estes sistemas, segundo Smith (2016), podem ser considerados também como dispositivos informatizados por possuírem capacidade de processamento. Ele ainda reforça que a tecnologia presente nos automóveis está tendendo mais à complexidade e à conectividade. Essa tendência atrelada à

conectividade ressaltada pelo autor vai ao encontro do conceito de Internet das Coisas (*Internet of Things – IoT*), dando potencialidade a estudos de aplicações explorando o tema. Segundo o projeto *Coordination And Support Action for Global RFID-related Activities and Standardisation* (Casagras, 2011), o conceito de Internet das Coisas se refere a uma infraestrutura de rede global capaz de interligar objetos físicos e virtuais através da exploração da capacidade de capturar de dados e de se comunicarem entre si.

1.2 MOTIVAÇÃO

Existe uma complexidade nos sistemas presentes nos automóveis, e isso se justifica com a evolução tecnológica e a informatização automotiva. Devido a este fator dominante, a percepção de algum eventual defeito em qualquer um desses conjuntos eletrônicos do veículo é uma tarefa complexa e dificilmente visível ao condutor.

Baseado nestas informações, é importante explorar estes conceitos para entender a relação desses sistemas embarcados automotivos com a informática, e como a tecnologia da informação pode contribuir para alavancar o crescimento desta área, podendo proporcionar conhecimento amplo destes sistemas, desde a arquitetura de operação, comunicação e processamento de dados veiculares, a fim de facilitar o entendimento de possíveis defeitos ou anomalias eletrônicas, além de estudar propostas de monitoramento e diagnóstico que podem facilitar o entendimento destes sistemas.

1.3 DEFINIÇÃO DO PROBLEMA

Analizando os fatos apresentados, identifica-se que o condutor está cada vez mais distante de entender o funcionamento do automóvel e consequentemente seus eventuais problemas eletrônicos que estão sujeitos a apresentarem. Esse distanciamento é justificado parcialmente devido ao sistema apresentar alta complexidade, e também pelo fato dos problemas não emitirem sinais aparentes sinalizando algum defeito. Essa distância é aumentada à medida que o gerenciamento do automóvel vai se imergindo na informática, automatizando parte de seu funcionamento com o uso de sensores e sistemas embarcados.

1.4 ABORDAGEM PROPOSTA

Este trabalho propõe a exploração e estudo direcionado ao funcionamento da arquitetura e comunicação dos sistemas embarcados presentes nos automóveis. A partir deste estudo, o objetivo será implementar um software embarcado que será responsável por interagir com a rede veicular interna utilizando um computador de baixo custo, seguindo a tendência de uma aplicação para *IoT*.

1.5 TRABALHOS RELACIONADOS

Foram pesquisados alguns trabalhos com propostas semelhantes a fim de entender algumas dificuldades enfrentadas e propor algumas melhorias. De todos os trabalhos coletados, foram analisadas três propostas relacionadas ao tema deste.

A primeira obra analisada foi proposta por Marques (2004), que apresentava o desenvolvimento de um sistema composto por hardware e software em tempo real para a comunicação com a rede *CAN* dos sistemas automotivos. O trabalho é voltado para análise e diagnóstico destes sistemas veiculares.

O segundo trabalho foi apresentado por Fagundes, Silva & Assis (2015), que se propuseram a coletar os dados da interface *OBD-II* e trata-los utilizando um microcontrolador, para assim poder enviar essas informações utilizando uma rede GSM. Estas informações seriam lidas através de um browser de internet em um computador.

O terceiro trabalho foi proposto por Staroski (2016), cujo objetivo é estudar a viabilidade de desenvolvimento de um protótipo de software embarcado em uma placa de *Raspberry Pi* para monitorar os sensores presentes em um automóvel. Com este protótipo seria possível o monitoramento via web em tempo real do veículo.

O trabalho apresentado por Staroski (2016) apresenta alguns pontos que podem ser melhorados e que serão explorados durante a elaboração desta proposta.

2 JUSTIFICATIVA

De acordo com o que foi abordado na introdução, sabe-se que o desgaste das peças de um automóvel é natural e inevitável. Entretanto, ter a possibilidade de prever o desgaste fazendo uma análise geral do estado de cada item que compõe o veículo pode ser uma alternativa satisfatória. Quando alguma peça começa a apresentar algum defeito e não é tratado com o devido cuidado, pode acarretar em problemas maiores, levando o condutor a ter gastos excessivos com a manutenção.

A evolução da tecnologia no cenário automobilístico traz várias melhorias, visando ao conforto e segurança do condutor, confiabilidade nos sistemas e otimização de consumo. Entretanto, por conter certa complexidade, acaba dificultando a percepção de algum desgaste ou falha de alguma peça ou dispositivo específico. Percebe-se aqui o resultado da informatização dos sistemas automotores. A indústria automotiva, segundo Smith (2016), tem criado veículos com sistemas eletrônicos de alta complexidade, mas disponibilizou poucas informações sobre como esses sistemas funcionam.

Percebe-se aqui uma abstração de funcionamento muito grande que ocorre dentro dos sistemas embarcados veiculares. Essa abstração dificulta a detecção de falhas justamente por não emitir sinais aparentes. O autor também reforça que normalmente esses sistemas eletrônicos automotivos são normalmente fechados, com exceção somente para a oficina mecânica da concessionária.

Voltando à contextualização sobre o fato da imersão automobilística no mundo informatizado, segundo o relatório de Charette (2009), publicado no *IEEE Spectrum*, ele observa que existem de 70 a 100 microprocessadores integrados em unidades de controle eletrônico (*electronic control units – ECUs*) e que são capazes de executar cerca de 100 milhões de linhas de código de software. Smith (2016) ainda reforça afirmando que à medida que os sistemas informáticos se tornam mais integrantes dos veículos, a realização de avaliações de segurança torna-se mais importante e complexa.

Baseado nessas informações e considerando a tendência tecnológica dos veículos, é importante estudar e aplicar os conceitos de tecnologias de baixo custo para a interação com a rede automotiva além das possíveis aplicações envolvendo a Internet das Coisas para auxiliar no diagnóstico e monitoramento automotivo.

3 OBJETIVOS

Esta seção abordará os objetivos gerais e específicos deste trabalho.

3.1 OBJETIVO GERAL

Este trabalho visa buscar um conhecimento amplo de toda a arquitetura automotiva relacionada aos seus sistemas embarcados, e como esses sistemas se comunicam entre si. Além deste estudo, o trabalho propõe também: buscar conhecimento técnico sobre sistemas computadorizados de baixo custo, com a finalidade de poder interagir com a rede veicular interna; adquirir aprendizado sobre a arquitetura, a infraestrutura e o armazenamento web utilizando serviços de computação em nuvem, como a *Amazon Web Services (AWS)*, a fim de manter os dados que foram coletados e enviados através de um dispositivo de baixo custo.

Desta forma, este trabalho propõe a implementação de um sistema computadorizado de baixo custo que se conecte e interaja com a rede interna do automóvel – sendo possível coletar algumas informações presentes nesta rede – e que transmita os dados para um serviço de computação em nuvem com o objetivo de manter essas informações para uma futura análise e ou alguma aplicação envolvendo a Internet das Coisas.

3.2 OBJETIVOS ESPECÍFICOS

Para alcançar o sucesso deste trabalho, o objetivo geral foi dividido em alguns objetivos específicos que estão listados abaixo:

- Estudar a arquitetura dos sistemas embarcados presentes nos automóveis;
- Levantar e estudar os protocolos de comunicação utilizados pela rede interna automotiva;
- Pesquisar por sistemas computadorizados de baixo custo e estudo da arquitetura;
- Analisar a viabilidade de integração de um sistema computadorizado de baixo custo à rede veicular interna;
- Desenvolver um software embarcado responsável por realizar a conexão e interação do dispositivo computadorizado com a rede interna do automóvel;
- Estudar a arquitetura e infraestrutura dos serviços de computação em nuvem;
- Implementar um banco de dados utilizando uma infraestrutura de computação em nuvem;
- Integrar o software com o banco de dados para armazenar as informações coletadas;

- Desenvolver uma página web para consultar e disponibilizar as informações contidas no banco de dados.

4 FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão abordados todos os assuntos que foram estudados para a construção deste trabalho.

4.1 SISTEMAS E SOFTWARE EMBARCADOS

A definição de sistemas embarcados, segundo Lee & Seshia (2017), são sistemas computacionais pouco perceptíveis, que geralmente são responsáveis por executar pequenas atividades de forma autônoma, como controlar os robôs da linha de produção de uma fábrica ou gerenciar os semáforos de uma cidade. De acordo com Carro & Wagner (2003) os sistemas computacionais embarcados estão presentes em boa parte das atividades humanas, passando desde o sistema de transporte até os eletrodomésticos de uma residência. Baseado nestes argumentos, sistemas embarcados são todos os dispositivos com poder de processamento, memória e fontes de energia limitados (Lee; Seshia, 2017), que podem se integrar com o meio físico.

Lee & Seshia (2017) ainda reforçam que os programas que são executados nestes dispositivos são chamados de software embarcado. Gill (apud Lee; Seshia, 2017) da *National Science Foundation in the US* cria o termo sistemas ciberfísicos (*Cyber-Physical Systems – CPS*) para se referir à integração da computação com processos físicos. Observa-se aqui uma outra definição para sistemas embarcados. No *CPS*, os sistemas informatizados monitoram e controlam os processos físicos geralmente executando instruções dentro de loops. Lee & Seshia (2017) ainda reforça que é importante compreender a dinâmica dos sistemas computacionais junto com os processos físicos. Por lidar diretamente com o mundo físico, o tempo necessário para executar uma tarefa, nos sistemas ciberfísicos, pode ser fundamental para o correto funcionamento do sistema. A passagem do tempo no mundo físico é algo crítico, ao contrário do mundo cibernetico.

Enquanto no processo físico existem diversas coisas acontecendo concorrentemente (ao mesmo tempo), nos processos de software as atividades acontecem em etapas sequenciais. Lee & Seshia (2017) afirmam que o maior desafio técnico na concepção e análise do software embarcado se deriva da necessidade de unir a semântica sequencial do mundo lógico com a realidade concorrente do mundo físico.

4.2 SISTEMAS DISTRIBUÍDOS

Um sistema distribuído, segundo Tanenbaum (2007), é um conjunto de computadores independentes que se apresentam ao usuário final como um único sistema coerente. De maneira genérica, a arquitetura de um sistema distribuído é composto por diversos itens de hardware que atuam de forma autônoma - geralmente processando informações específicas - mas que se comunicam entre si caracterizando-se como apenas um único hardware responsável

pelo sistema como um todo. O autor ainda reforça que as pessoas ou usuários acham que estão interagindo com um sistema apenas, e não com um conjunto de sistemas. Contudo, para garantir a operação de todo este conjunto, é preciso que haja a colaboração de todos os componentes e dispositivos que fazem parte do sistema. A essência do sistema distribuído está em estabelecer essa colaboração.

Não é estabelecido nenhum padrão ou premissa relacionado ao tipo de hardware que irá compor o sistema. Tanenbaum (2007) ainda afirma que esses dispositivos podem variar desde computadores centrais até pequenos nós em redes de sensores. Também não existe premissa com relação ao modo com que os dispositivos se interligam. A característica principal dos sistemas distribuídos está em ocultar boa parte da comunicação interna deste conjunto ao usuário. Tanenbaum (2007) define ainda quatro metas que devem ser cumpridas para que o esforço necessário para a construção de um sistema distribuído seja válida: o sistema deve oferecer fácil acesso a seus recursos; deve ocultar razoavelmente bem o fato de que os recursos são distribuídos por uma rede; deve ser aberto e deve permitir a sua expansão.

4.3 ARQUITETURA DO SISTEMA AUTOMOTIVO

Para compreender como funciona a comunicação interna do veículo, é necessário antes saber como o sistema está estruturado, quais dispositivos e controladores utilizados, qual a arquitetura adotada e quais protocolos estão implementados. Para Navet & Simonot-Lion (2008), os fabricantes de automóveis diferenciam em várias categorias os eletrônicos embarcados que um carro possui. Eles utilizam essas categorias para agrupar sistemas mecânicos e ou eletrônicos de acordo com as suas funcionalidades.

4.3.1 CATEGORIAS FUNCIONAIS DOS SISTEMAS EMBARCADOS AUTOMOTIVOS

Historicamente, segundo os autores, existem cinco categorias de sistemas embarcados: *Power Train*, *Chassis*, *Body*, *HMI* e *Telematics*. A categoria *Power Train* fazem parte todos os sistemas que participam da propulsão longitudinal do veículo, incluindo o motor, a transmissão e todos os componentes que dão apoio para esta função. A categoria *Chassis* se refere às quatro rodas e à sua posição relativa de movimento. Nesta categoria os principais sistemas são o de freios e direção. Dentro da categoria *Body* estão presentes as entidades que não pertencem à dinâmica do veículo, mas que auxiliam o motorista, como o airbag, limpadores, iluminação, vidros, ar condicionado, assentos, etc. Já a categoria *HMI* inclui o equipamento que permite a troca de informações entre os sistemas eletrônicos e o motorista do veículo. Por fim, a categoria *Telematics* está relacionado à componentes que permitem a troca de informações do veículo com o mundo exterior, como rádio, sistemas de navegação, GPS, entre outros. Navet & Simonot-Lion (2008) ainda ressaltam que cada categoria do sistema eletrônico possui características bem

diferentes, o que faz com que cada dispositivo tenha um requisito ou uma restrição bem definida. A Figura 1 ilustra a divisão funcional dos sistemas pelas categorias.

Figura 1 – Representação das cinco categorias funcionais.



Fonte: baseado em Navet & Simonot-Lion (2008)

- Categoría *Power Train*: Além de controlar a velocidade do motor, atuando de acordo com as intervenções do motorista no pedal, o controlador pode também, atuar de acordo com fatores naturais, como a temperatura do ar ou o nível de oxigênio, ou atuar de acordo com os distúrbios ambientais, como a poluição dos gases de escape ou o ruído. Ainda segundo Navet & Simonot-Lion (2008), o controlador é projetado para otimizar alguns parâmetros. O parâmetro mais comum a ser controlado é a quantidade de combustível que deve ser injetado para combustão de acordo com a rotação do motor e a posição do pedal do acelerador. Observa-se que nesta categoria estão presentes todos os dispositivos que auxiliam no gerenciamento do motor, tanto de forma direta quanto indireta.
- Categoría *Chassis*: Nesta categoria, existem sistemas responsáveis por gerenciar a interação do veículo com a estrada. O objetivo destes sistemas é controlar o automóvel de acordo com as solicitações do motorista, como frenagem ou aceleração, considerando também o perfil da via ou as condições ambientais, visando sempre o conforto e a segurança dos passageiros. Navet & Simonot-Lion (2008) ainda reforça que esses sistemas devem ser de

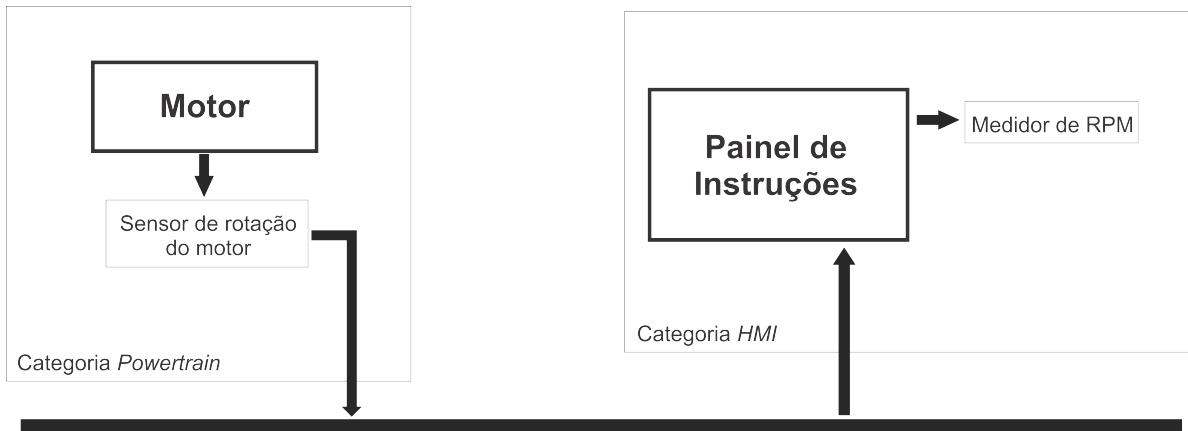
alta qualidade, como qualquer sistema crítico. Os sistemas mais comuns são o de frenagem (ABS) e o controle automático de estabilidade (ASC).

- Categoría *Body*: Limpadores de para-brisa, luzes, portas e janelas e outros itens são controlados por sistemas pertencentes a esta categoria. Estes, por sua vez, não estão sujeitos a restrições de desempenho rigorosas, e do ponto de vista de segurança, segundo os autores, não representam uma parte crítica do sistema. Entretanto, existem certas funções, como controlar o acesso ao veículo, que deve respeitar as dificuldades em tempo real.
- Categoría *HMI*: Os sistemas presentes nesta categoria, em geral, permite a interação do motorista com diversas funções integradas no veículo. Uma das funções é exibir o estado atual do veículo, como velocidade, rotação do motor e temperatura, por exemplo, ou também mostrar o estado de algum dispositivo multimídia.
- Categoría *Telematics*: Esta categoria, ainda segundo Navet & Simonot-Lion (2008), inclui sistemas que suportam trocas de informações entre infra-estruturas viárias e rodoviárias. Um exemplo está relacionado com a cobrança automática de pedágios. Para os autores, em um futuro próximo, esta categoria permitirá otimizar o uso rodoviário através da gestão do tráfego a fim de evitar congestionamentos.

A divisão dos sistemas por categoria engloba dispositivos de acordo com suas semelhanças funcionais. Logo, se um conjunto eletrônico, mesmo que sejam diferentes, seguirem um determinado requisito comum à uma categoria, estes pertencerão à esta. Entretanto, os sistemas eletrônicos, mesmo pertencendo à uma categoria distinta, não são impedidos de se comunicarem entre si. Segundo o exemplo de Navet & Simonot-Lion (2008), as informações como a rotação do motor, temperatura ou a velocidade fazem parte do gerenciamento do motor, e logo pertencem à categoria *power train*, mas são transmitidas desta para a categoria *HMI*, para exibir as informações do veículo ao motorista no painel de instruções (Figura 2).

Analizando o funcionamento destes sistemas eletrônicos, eles são caracterizados como sistemas embarcados automotivos pelo fato de possuírem uma forte interação com o mundo físico, conforme a definição de sistemas embarcados explorado na seção 4.1. Para interagir com o mundo físico, estes sistemas fazem o uso de outros dispositivos conhecidos como sensores e atuadores. Segundo Lee & Seshia (2017), um sensor mede uma quantidade física, enquanto o atuador altera a quantidade física. Eles ainda complementam que estes dispositivos conectam o mundo cibernetico com o mundo físico. Em outras palavras, um sensor obtém os dados de uma leitura, e o atuador realiza uma ação que foi passada a ele. Em um automóvel existem diversos sensores espalhados por sua estrutura e que monitoram diversos aspectos físicos do veículo, como aceleração lateral, velocidade e tração individual das rodas, monitorando a estabilidade

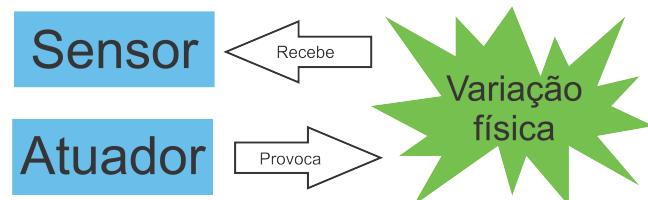
Figura 2 – Representação do monitoramento de RPM e a comunicação entre as categorias.



Fonte: baseado em Navet & Simonot-Lion (2008)

do carro durante o percurso (Navet; Simonot-Lion, 2008). O autor ainda complementa que quando uma correção precisa ser aplicada nesta situação, as rodas dianteiras ou traseiras podem frear individualmente e com intensidades diferentes, ou atuar na redução da potência do motor. Percebe-se neste exemplo a ação dos atuadores no meio físico. A Figura 3 representa a influência destes dispositivos no meio físico.

Figura 3 – Representação da atuação dos sensores e atuadores.



Fonte: baseado em Lee & Seshia (2017)

Analizando-se também a forma como esses dispositivos trocam informações uns com os outros, percebe-se que está presente o conceito de sistemas distribuídos, conforme apresentado na seção 4.2. Um exemplo desse conceito no cenário automotivo é mencionado por Navet & Simonot-Lion (2008), que está relacionado com a funcionalidade de piloto automático (*cruise control*) presente em alguns modelos de veículos. Para que esse sistema funcione perfeitamente, segundo os autores, é necessário a troca de dados de vários sensores que podem pertencer a categorias funcionais distintas. A função de piloto automático tem a responsabilidade central de processar esses dados vindos de diversas origens e enviar as respostas a vários outros dispositivos de saída para cumprir seu objetivo.

Desde a década de 70, como aponta Navet & Simonot-Lion (2008), houve um aumento muito grande no número de sistemas que passaram a substituir os que eram puramente mecânicos ou hidráulicos. O desempenho e confiabilidade desses componentes de hardware que passaram a integrar o automóvel permite a execução de funções complexas, aumentando o conforto e

a segurança dos ocupantes do veículo. A arquitetura de hardware de um automóvel não é composta somente de sensores, atuadores, controladores e links de comunicação que permite a interconexão dos componentes, mas também de dispositivos conhecidos como *ECUs*.

4.3.2 ECU (ELECTRONIC CONTROL UNIT)

Um veículo possui alguns controladores eletrônicos, que para Smith (2016), são chamados de dispositivos informatizados que passam por diversos nomes diferentes, como Unidade de Controle Eletrônico (*Electronic Control Unit - ECU*), Unidade de Controle do Motor (*Engine Control Unit – ECU*), Unidade de Controle de Transmissão (*Transmission Control Unit – TCU*) ou ainda Módulo de Controle de Transmissão (*Transmission Control Module – TCM*). O autor ainda destaca que esses termos na teoria podem ter significados específicos em uma determinada configuração, mas na prática acaba utilizando-se o termo *ECU* que é comum a eles. Isso porque independente do tipo de controlador eletrônico, eles acabam executando as mesmas funções, ou funções extremamente semelhantes. Navet & Simonot-Lion (2008) afirmam que um dos principais propósitos dos sistemas eletrônicos é auxiliar o condutor a controlar o veículo. Segundo eles, a *ECU* autônoma é um subsistema composto por um microcontrolador e um conjunto de sensores e atuadores associados. Os autores também trazem a ideia de que as *ECUs* podem aprimorar a ação dos atuadores, indo muito além da capacidade humana sobre eles. A Figura 4 apresenta a foto de uma *ECU*.

Figura 4 – Foto de uma *ECU*.

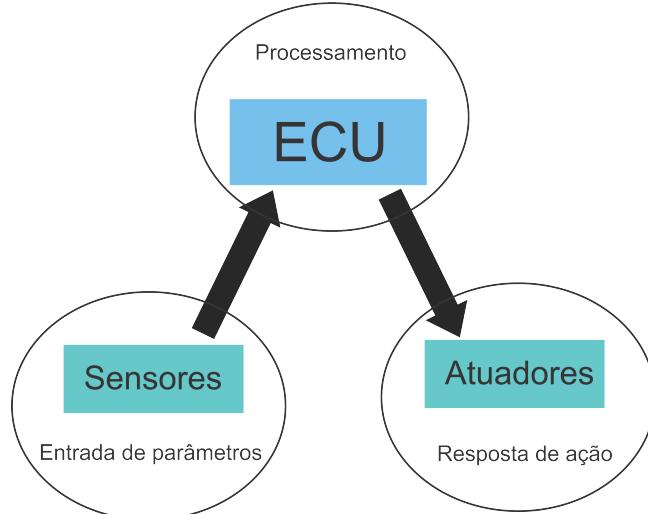


Fonte: imagem extraída do site <http://carrosinfoco.com.br/wp-content/uploads/2012/07/ecu1.jpg>

Em outras palavras, a *ECU*, de forma geral, é um dispositivo capaz de processar as informações recebidas pelos sensores dos veículos, e gerar uma resposta para a ação dos atuadores, conforme está representado na Figura 5. Como a *ECU* faz parte de um sistema embarcado, sua memória e capacidade de processamento são limitadas assim como qualquer dispositivo embarcado, conforme abordado na seção 4.1. Entretanto, as *ECUs* são destinadas a

executar determinadas tarefas em específico, de modo que o desempenho do sistema como um todo não seja afetado pela limitação de hardware do dispositivo.

Figura 5 – Diagrama da comunicação entre *ECU*, sensor e atuador.



Fonte: baseado em Navet & Simonot-Lion (2008)

Segundo Smith, a comunicação da *ECU* com os diversos componentes acontece de forma simplificada. Para esta comunicação ser simples e eficiente, os automóveis utilizam alguns protocolos para tratar da comunicação da rede interna do veículo. Estes protocolos serão explorados na sequência.

4.3.3 PROTOCOLOS AUTOMOTIVOS

Apesar dos protocolos permitirem a comunicação entre os dispositivos, Smith (2016) reforça que caso o veículo tenha sido fabricado antes do ano de 2000, há possibilidade de ele não possuir nenhum protocolo implementado. Para ele, os protocolos presentes nos barramentos são responsáveis por gerenciar a transmissão de pacotes de dados pela rede automotiva. Diversos sensores e dispositivos conectados nesta rede se comunicam com a finalidade de administrar o comportamento do veículo. Segundo o autor, toda a parte da comunicação crítica ocorrem nos barramentos de alta velocidade, enquanto a comunicação não crítica ocorre nos barramentos de média e baixa velocidade. Subentende-se que a comunicação crítica está relacionada a todos os controles que garantem a segurança e integridade dos ocupantes do automóvel.

Navet & Simonot-Lion (2008) afirmam que o papel central das redes está em manter os sistemas embarcados em estado de segurança, uma vez que a maioria das funções críticas estão distribuídas e necessitam da comunicação contínua entre si. Ele ainda aponta que a diversificação das redes utilizadas em todo o automóvel é justificada pela crescente necessidade da largura de banda, desempenho e outros requisitos de confiabilidade.

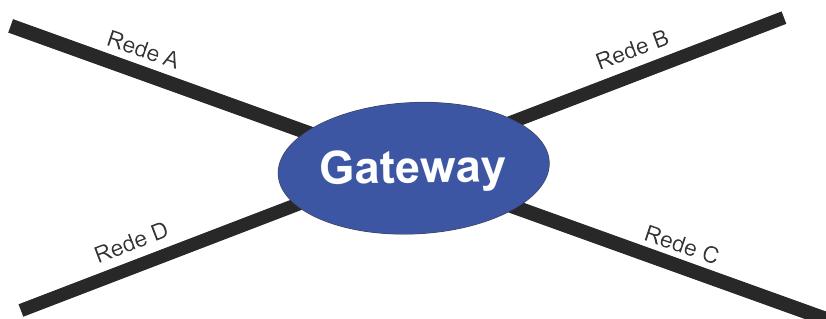
Cada fabricante decide qual barramento ou protocolo serão utilizados na arquitetura do veículo; entretanto, existe um protocolo comum em todos os veículos: o protocolo CAN (Smith, 2016). Em 1994, a Sociedade de Engenheiros Automotivos (*Society for Automotive Engineers – SAE*), citada por Navet & Simonot-Lion (2008), definiu uma classificação para protocolos de comunicação automotiva baseando-se na velocidade de transmissão dos dados e nas funções que seriam distribuídas pela rede (Tabela 1). As redes de classe A possuem uma taxa de dados inferior a 10 kbps e são utilizadas para transmissão de dados de controle com tecnologias de baixo custo e estão integradas na categoria *Body*. As redes classe B operam em uma velocidade de 10 a 125 kbps e se dedicam para suportar a troca de dados entre *ECUs*, para reduzir o número de sensores compartilhando informações. Aplicações que precisam de comunicação em tempo real requerem redes de classe C (com velocidade de transmissão de 125 kbps a 1Mbps) ou redes de classe D (operando com velocidades superiores a 1Mbps). As redes classe C integram a comunicação das categorias *Power Train* e *Chassis*. Já as redes de classe D são dedicadas a dados multimídia e aplicações críticas de segurança que requeiram previsibilidade e tolerância de falhas.

Tabela 1 – Tipos de redes e suas respectivas velocidades de transmissão de acordo com a SAE.

Tipos de redes	Velocidade de transmissão
Classe A	até 10 kbps
Classe B	de 10 a 125 kbps
Classe C	de 125 kbps a 1Mbps
Classe D	acima de 1Mbps

De acordo com Navet & Simonot-Lion (2008), é comum a inclusão dos quatro tipos de redes em barramentos diferentes interligadas por *gateways* na arquitetura eletrônica dos veículos atuais, conforme representado na Figura 6. Eles ainda complementam que será possível futuramente a inclusão de um barramento dedicado aos sistemas de segurança dos ocupantes. De todos os protocolos existentes e citados pelos autores, serão explorados dois, o protocolo CAN e o SAE J1850.

Figura 6 – Representação de redes distintas interligadas.



Fonte: baseado em Navet & Simonot-Lion (2008)

4.3.3.1 PROTOCOLO CAN (*CONTROLLER AREA NETWORK*)

Por volta de 1980, foi desenvolvido pela Bosch o protocolo conhecido como *CAN* (*Controller Area Network*), ou simplesmente protocolo de controle de área de rede. Este protocolo foi integrado pela primeira vez em carros de produção da Mercedes na década de 90 (Navet; Simonot-Lion, 2008). Smith (2016) afirma que o *CAN* é um protocolo simples utilizado na indústria automobilística. Este protocolo permite a comunicação de *ECUs* e sistemas embarcados que estão presentes nos veículos modernos. Nivet & Simonot-Lion (2008) ainda complementam que as redes que utilizam esse protocolo tornaram-se as mais utilizadas nos sistemas automotivos. Por possuírem baixo custo, serem robustas e terem atrasos baixos de comunicação, o protocolo *CAN* é um padrão utilizado na Europa para transmissão de dados de aplicações automotivas.

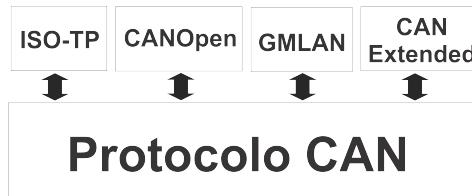
Atualmente, para controlar os sistemas da categoria *Power Train* utiliza-se a *CAN* como uma rede SAE de classe C, operando de 250 a 500 kbps. Entretanto, ela também pode operar nos sistemas de categoria *Body*, com uma taxa de 125 kbps. Segundo os estudos de Smith (2016), o protocolo *CAN* possui dois tipos de pacotes que são chamados de *standard* (padrão) e *extended* (extendido).

Os pacotes *standard* possuem quatro elementos, que são o ID de arbitragem, extensão de ID, código do tamanho dos dados e os próprios dados. Quando um dispositivo tenta se comunicar é enviado ID de arbitragem, que é uma mensagem de broadcast que identifica o ID deste dispositivo. Quando dois pacotes são transmitidos ao mesmo tempo no barramento, aquele com ID de arbitragem menor vence. Nivet & Simonot-Lion (2008) apontam esta técnica como sendo um método utilizado por este protocolo para evitar colisões ao acessar o barramento. Voltando aos estudos de Smith (2016), ele afirma que o bit que define a extensão de ID tem o valor 0 para pacotes *standard*. O código do tamanho dos dados define o tamanho dos dados, que pode variar de 0 a 8 bytes. O tamanho máximo dos dados transportados por este pacote pode ter no máximo 8 bytes.

Já os pacotes *extended*, segundo o autor, são semelhantes ao *standard*, com exceção de possuir espaço maior para armazenar IDs mais longos. Estes pacotes foram desenvolvidos para caberem dentro do formato *standard* para manter a compatibilidade com as versões anteriores. Desta forma, caso algum sensor tenha suporte somente para pacotes *standard*, ele não será invalidado caso sejam transmitidos pacotes *extended* na mesma rede. Dentre outras particularidades deste pacote, Smith (2016) ainda reforça que existem outros protocolos adicionais específicos de alguns fabricantes (protocolo ISO-TP, *CANopen*, *GMLAN*) que seguem o padrão *CAN*, da mesma forma que o pacote *CAN extended* (Figura 7).

Segundo Nivet & Simonot-Lion (2008), o protocolo *CAN* define apenas a camada física e a camada de link dos dados. Para tratar a padronização de inicialização, implementar a

Figura 7 – Representação de outros protocolos implementados no padrão CAN.



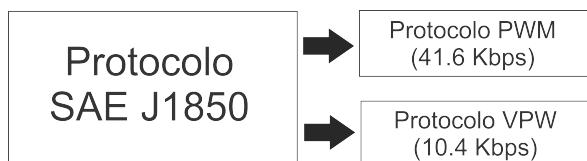
Fonte: baseado em Smith (2016)

segmentação de dados ou enviar mensagens periódicas, foram propostos diversos protocolos de nível superior.

4.3.3.2 PROTOCOLO SAE J1850

O protocolo SAE J1850 foi implementado originalmente por volta de 1994, segundo Smith (2016), e ainda pode ser encontrado nos veículos atuais. Comparado com o protocolo CAN, este é mais lento, entretanto, seu custo de implantação é mais barato. Navet & Simonot-Lion (2008) menciona que são definidas duas variações para este protocolo. Smith (2016) as define como modulação de largura de pulso (*pulse width modulation – PWM*) e largura de pulso variável (*variable pulse width – VPW*), conforme representado pela Figura 8. O *PWM* trabalha com uma velocidade de 41.6 Kbps, enquanto o *VPW* trabalha com 10.4 Kbps.

Figura 8 – Variações do protocolo SAE J1850.



Fonte: baseado em Smith (2016), Navet & Simonot-Lion (2008)

Navet & Simonot-Lion (2008) afirmam que para tratar dos controles dos sistemas da categoria *Body* ou de diagnósticos, os Estados Unidos adotaram este protocolo SAE J1850 de classe B pois as comunicações não possuíam requisitos de transmissão em tempo real.

Analizando os protocolos apresentados, observa-se que para determinada aplicação é utilizado um protocolo específico. E cada protocolo tem uma classificação quanto à sua velocidade de transmissão pelo SAE. Existem outros protocolos que estão presentes na rede automotiva, como o *Keyword Protocol* e ISO 9141-2, explorados por Smith (2016), os protocolos IDB-1394, VAN (*Vehicle Area Network*) e TTCAN, explorados por Navet & Simonot-Lion (2008), ou ainda os protocolos LIN (*Local Interconnect Protocol*), MOST (*Media Oriented Systems Transport*) e o FlexRay, todos explorados na literatura de ambos.

Observando-se o comportamento da rede automotiva, nota-se a presença de vários protocolos responsáveis pela comunicação de diversos dispositivos que são destinados a realizar

uma determinada função específica no veículo. Entretanto, também existe a necessidade de monitorar essa rede para realizar eventuais diagnósticos neste sistema, e o nome dado para esta função se chama diagnóstico de bordo (*OnBoard Diagnostics – OBD*) (Navet; Simonot-Lion, 2008).

4.3.4 CONECTOR *OBD-II* (*ONBOARD DIAGNOSTICS*)

Também conhecido como conector de diagnóstico de conexão (*Diagnostic Link Connector – DLC*), o conector *OBD-II* está integrado em grande parte dos veículos, e permite a comunicação com a rede interna automotiva conforme exemplificado por Smith (2016). De acordo com Navet & Simonot-Lion (2008), a introdução de sistemas informáticos capazes de memorizar grandes quantidades de informação de um automóvel possibilitou o diagnóstico de bordo, que se refere ao autodiagnóstico e a facilidade de emissão de relatórios. Para Smith (2016), o conector *OBD-II* é utilizado principalmente pela mecânica para analisar e solucionar eventuais problemas com o automóvel. A Figura 9 mostra a localização do conector *OBD-II* em um automóvel.

Figura 9 – Foto do conector *OBD-II*.

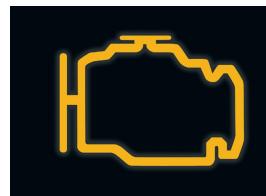


Fonte: foto tirada pelo autor

Ao apresentar uma falha, o sistema guarda informações relacionadas a ela e aciona uma luz de aviso do motor no painel do motorista, conhecida como luz indicadora de mau funcionamento (*malfunction indicator lamp – MIL*, Figura 10). Embora o diagnóstico tenha se limitado a essa luz indicadora, conforme Navet & Simonot-Lion (2008) apresentam, a comunicação dos sistemas *OBD* recentes são padronizados, como por exemplo a padronização de dados monitorados, codificação padronizada e relatórios de uma lista de falhas específicas, conhecidas como códigos de diagnósticos de problemas (*Diagnostic Trouble Codes – DTC*). De acordo com Smith (2016), as verificações de rotina são tratadas pela *ECU* primária do veículo, o qual ele se refere de módulo de controle do *powertrain* (*PCM*). Para controlar a emissão de gases de escape ao longo da vida

útil de um veículo, segundo Navet & Simonot-Lion (2008), foi necessário uma padronização na especificação *OBD-II*, que passou a ser obrigatória para todos os carros comercializados nos Estados Unidos a partir de 1996. Este padrão definia com precisão diversos aspectos relacionados ao diagnóstico, avaliação e monitoramento do veículo.

Figura 10 – Foto da luz *MIL* do painel.

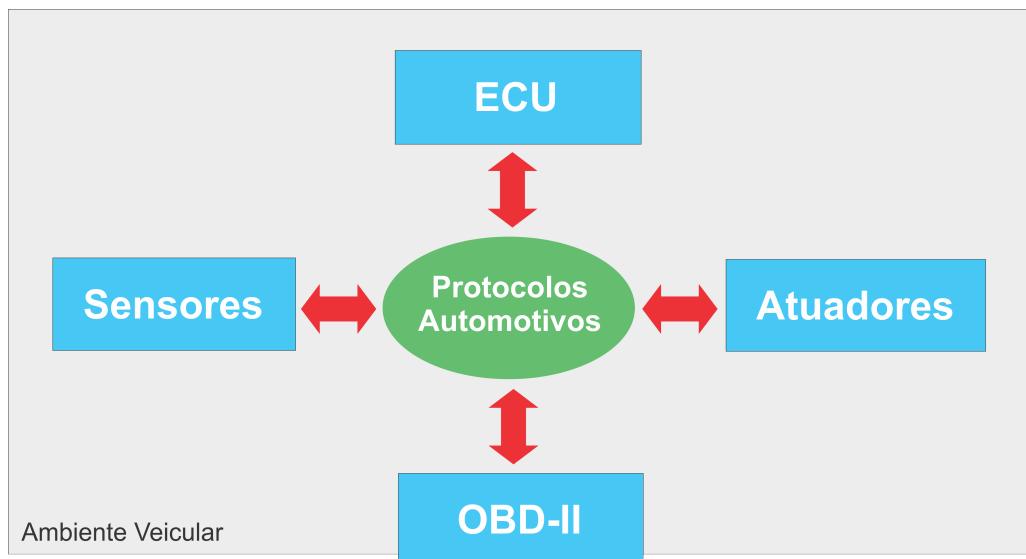


Fonte: imagem extraída do site <https://goo.gl/CiSaec>

Voltando à parte de diagnósticos, com base nos estudos de Smith (2016), todos os códigos de falha, como os *DTCs* são armazenados no *PCM*. Ele ainda complementa dizendo que os *DTCs* são armazenados em locais diferentes. Enquanto os *DTCs* baseados em memória ficam armazenadas na memória RAM e apagados quando a energia da bateria acaba, os *DTCs* mais sérios que estão relacionados com falhas permanentes ficam armazenados em locais onde a persistência de dados é maior e consequentemente sobreviverão à uma queda de energia.

A Figura 11 representa de maneira simplificada como ocorre a comunicação entre os dispositivos na rede interna automotiva.

Figura 11 – Diagrama representando a arquitetura da rede veicular.



Fonte: baseado em Navet & Simonot-Lion (2008) e Smith (2016)

4.4 ELM327

A lei atualmente obriga, segundo a ELM Electronics (2012), que todos os automóveis produzidos forneçam uma interface para a conexão de dispositivos de teste e diagnóstico. Esta

interface apresentada na subseção 4.3.4 como *OBD-II*, segue as mesmas especificações dos protocolos da rede interna automotiva, o que permite a interação com ela somente se utilizar os mesmos padrões que foram estabelecidos por esta rede. Entretanto, a ELM Electronics (2012) ressalta que tais padrões não são compatíveis com computadores ou outros dispositivos inteligíveis, como um smartphone, por exemplo. Para solucionar este problema e permitir a compatibilidade, o ELM327 foi desenvolvido para servir de intermediário entre a porta *OBD-II* - presente nos automóveis - com a interface serial padrão (RS232) presente nos computadores. A ELM Electronics (2012) ainda afirma que este dispositivo é capaz de identificar e interpretar nove protocolos automotivos mais o padrão J1939 utilizado por ônibus e caminhões. Desta maneira, com o ELM327 é possível utilizar um computador para se comunicar com a rede interna de um automóvel.

O ELM327 se conecta ao computador por meio da interface serial, que segundo a ELM Electronics (2012), esta interface pode ser virtualizada com adaptadores USB, ou por meio de dispositivos com suporte *bluetooth*. O fabricante reforça que o meio que o dispositivo usa para se conectar ao computador não tem muita relevância, pois para se comunicar com o veículo através da rede, basta apenas uma aplicação que faça o envio ou recebimento dos dados. Entretanto, para garantir a comunicação, alguns ajustes devem ser configurados, como a configuração da porta e a taxa de dados correta, conforme abordados no manual. A Figura 12 mostra como é um dispositivo ELM327 *Bluetooth*, e a Figura 13 expressa como este dispositivo se integra com o automóvel.

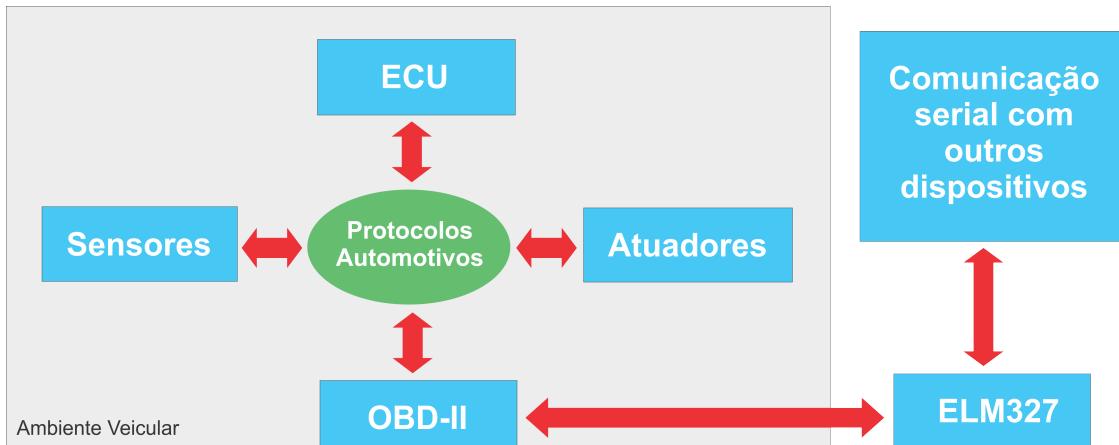
Figura 12 – Foto do ELM327 *Bluetooth*.



Fonte: foto tirada pelo autor

Existem dois tipos de comandos que podem ser enviados. Os comandos que se destinam ao próprio dispositivo, e os comandos destinados ao veículo. Segundo a ELM Electronics (2012), os comandos destinados ao dispositivo começam com os caracteres ‘AT’, enquanto os destinados à rede veicular contêm dígitos hexadecimais. Ele ainda ressalta que o ELM327 apenas converte

Figura 13 – Diagrama representando a interação do ELM327 com a rede automotiva.



Fonte: baseado em ELM Electronics (2012)

os protocolos, não realizando nenhum tipo de validação dos dados transmitidos. Entretanto, este dispositivo garante a entrega dos dados nas extremidades, ou seja, garante que os dados sejam transmitidos tanto para o computador quanto para o veículo.

4.4.1 COMANDOS AT

Como visto anteriormente, os comandos ‘AT’ são destinados ao próprio dispositivo ELM327. Existem vários parâmetros dentro dele que podem ser ajustados para modificar seu comportamento, segundo o fabricante. Esses comandos são semelhantes aos utilizados pelos modems para a configuração interna (ELM Electronics, 2012).

4.4.2 COMANDOS OBD

Quando um comando é enviado sem as iniciais ‘AT’, é entendido como comando *OBD* para o veículo. Desta forma, o dispositivo apenas verifica se o comando enviado é um algarismo hexadecimal ou uma sequência aleatória de caracteres, e logo após transmite o dado ao veículo. Para a ELM Electronics (2012), estes dados são empacotados e enviados ao automóvel. Os comandos enviados ao veículo necessitam geralmente de mais quatro bytes adicionais: três bytes de cabeçalho e um byte de verificação de erro, que devem ser incluídos junto aos dados que serão enviados. Contudo, o ELM327 adiciona estes bytes extras ao comando, abstraindo esta necessidade do usuário.

O comprimento dos comandos *OBD* pode variar de um a sete bytes, sendo este último o limite máximo aceito pelo dispositivo. Ao empacotar e enviar o comando pela interface *OBD*, o dispositivo fica monitorando o barramento veicular para obter a resposta ao comando. Obtida a resposta, ela será encaminhada para a porta serial ao usuário.

4.4.2.1 COMUNICAÇÃO COM O VEÍCULO - PADRÕES DE SOLICITAÇÃO

De acordo com a ELM Electronics (2012), o formato das solicitações que são enviadas ao veículo devem seguir um padrão definido. O primeiro byte enviado é denominado ‘modo’ (*mode*). Este byte informa que tipo de dados está sendo solicitado. O segundo byte é conhecido como ID de parâmetro, ou simplesmente número PID (*parameter identification*). Este especifica a informação real que é necessária, como acessar uma informação de um determinado dispositivo do automóvel, por exemplo. Os modos e PIDs são descritos em detalhes pelos padrões SAE J1979 ou ISO 15031-5, além de poderem ser definidos também pelos fabricantes dos automóveis. Entretanto, como menciona a ELM Electronics (2012), é comum um veículo não suportar todos os ‘modos’ e PIDs descritos pelo padrão. As mensagens de resposta, que geralmente são enviadas pela *ECU*, são retornadas em conjunto de números hexadecimais, havendo a necessidade de realizar algumas conversões para ter acesso aos valores que foram retornados. Uma segunda conversão é necessária de acordo com o PID que foi solicitado (a conversão do valor varia de acordo com o PID).

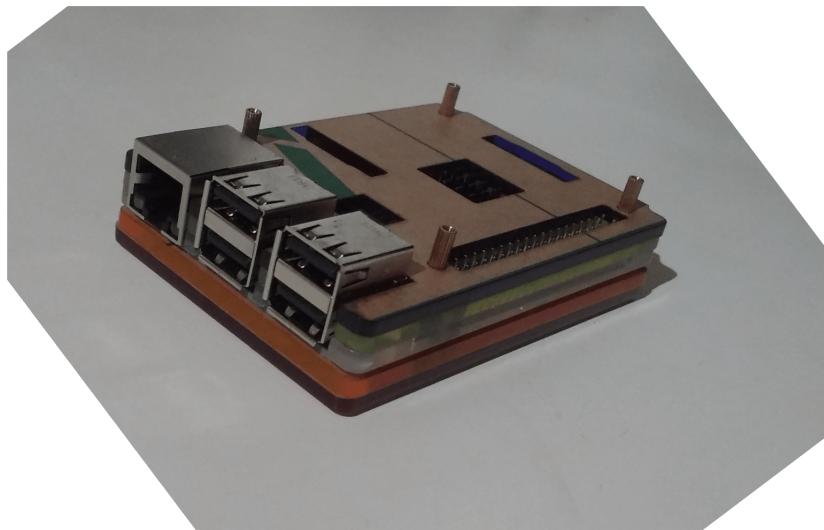
4.5 RASPBERRY PI

Segundo Oliveira (2013), o *Raspberry Pi* veio ao mercado com a proposta de ser um equipamento barato e dar suporte ao processo educacional das crianças. Segundo uma entrevista de Torvalds ao site *BBC News* em 2012, o projeto *Raspberry Pi* é algo importante, pois por ser de baixo custo, permite a exploração da informática sem a preocupação com possíveis danos ao hardware.

Richardson & Wallace (2013) afirmam que é possível realizar diversas atividades com o *Raspberry Pi*, como utilizá-lo para computação de maneira geral, aprender programação ou ainda integrar o minicomputador a projetos eletrônicos. Segundo os autores, um dos fatores que o diferenciam de um computador convencional além do seu tamanho e preço, é sua facilidade de integração com projetos eletrônicos. Apesar de parecer semelhante aos microcontroladores, as plataformas *System on a Chip (SoC)* possuem mais características em comum com um computador do que com um microcontrolador qualquer. A Figura 14 mostra a aparência física do *Raspberry Pi*.

O *Raspberry Pi*, segundo Oliveira (2013), é um computador montado em apenas uma única placa, o qual se classifica como *Single-Board Computer (SBC)*. Ele ainda afirma que nesta placa, são integrados o processador, memória, portas de I/O e outros componentes que são necessários para seu funcionamento. A arquitetura presente em seu processador é um ARM, semelhantes aos processadores utilizados em celulares, tablets ou sistemas embarcados. Richardson & Wallace (2013) afirmam que este processador é o mesmo que os encontrados no iPhone 3G e no *Kindle 2*, o que permite ter uma referência sobre seu poder de processamento. De acordo

Figura 14 – Foto do *Raspberry Pi 3*.



Fonte: foto tirada pelo autor

com os autores, os chips ARM possuem diferentes núcleos configurados para fornecer capacidades diferentes de processamento. Seu sistema operacional padrão é uma distribuição Linux conhecida como *Raspbian*, entretanto, os autores ressaltam que o usuário final não é limitado a utilizar somente este S.O. no dispositivo, ficando livre para explorar outras distribuições.

4.6 COMPUTAÇÃO EM NUVEM

De acordo com a definição da Microsoft (2017), computação em nuvem é a disponibilização de diversos serviços de computação - como servidores, armazenamento, banco de dados, sistemas, entre outros serviços de TI - através da internet. A *Amazon Web Service* ainda complementa que a plataforma que provê estes serviços é responsável pela manutenção de todo equipamento necessário para a disponibilização dos serviços. Tanto a *Amazon Web Services* (AWS) quanto a Microsoft *Azure* são plataformas conhecidas que proveem serviços de computação em nuvem.

Ambas afirmam que existem vários benefícios na utilização destes serviços. Dentre eles, uma das vantagens é relacionada ao custo, pois dispensa o gasto com qualquer tipo de equipamento para manter um datacenter local, uma vez que é utilizado uma infraestrutura já pronta e montada por estas prestadoras de serviço, que garantem a disponibilidade dos recursos através da internet (Amazon, 2017c; Microsoft, 2017).

Estes serviços são classificados em três tipos: Infraestrutura como Serviço (IaaS), Plataforma como Serviço (PaaS) e Software como Serviço (SaaS). A categoria de IaaS, segundo a definição da Microsoft (2017), é a mais básica e provém toda a infraestrutura de TI, como servidores, máquinas virtuais, armazenamento, redes e sistemas operacionais. A categoria PaaS são serviços de computação que fornecem ambientes sob demanda para desenvolvimento, teste,

fornecimento e gerenciamento de aplicativos de software. O SaaS fornece aplicativos de software pela nuvem sob demanda, normalmente baseadas em assinaturas.

4.7 INTERNET DAS COISAS (*INTERNET OF THINGS - IOT*)

A ideia de *IoT* se refere aos objetos do mundo físico gerando informações de forma autônoma para os computadores (Ashton, 2009). Ele reforça que todas as informações presentes na internet foram geradas a partir de um ser humano. Uma questão pontuada por ele é referida à limitação das pessoas com relação ao tempo, atenção e precisão. Essa limitação pode resultar em ineficiência ao capturar dados sobre o mundo real. Dito isso, o autor conclui afirmando que é necessário capacitar os computadores através de seus próprios meios de coleta de informações para observarem e compreenderem o mundo físico sem a limitação da entrada de dados através de uma pessoa.

O conceito de Internet das Coisas conforme definido pelo projeto Casagras (2011) na introdução e idealizado por Ashton (2009) abordam a interconectividade e troca de informações entre os objetos do mundo físico com o mundo virtual (Figura 15). Dias (2016) afirma que existem infinitas possibilidades de negócio e aplicações envolvendo o conceito de internet das coisas. Ela ainda menciona alguns nichos destes negócios, como os voltados para os bens de consumo, distribuição de energia, casas inteligentes, indústria e manufatura e transporte inteligente. Para este último nicho, a autora ainda traz algumas aplicações possíveis de serem exploradas, como notificação das condições de tráfego, controle inteligente de rotas, coordenação das rodovias e monitoramento remoto de veículos.

De acordo com o que foi apresentado até o momento, observam-se dois pontos importantes: primeiro, a cada evolução de um automóvel, nota-se que ele está se tornando mais informatizado e conectado; segundo, a tendência da *IoT* é permitir que cada vez mais as coisas se comuniquem e troquem informações entre si, com mínima intervenção humana.

Figura 15 – Representação da interação das 'coisas' dentro do conceito de *IoT*.



Fonte: baseado nas obras de Ashton (2009), Casagras (2011) e Dias (2016)

5 METODOLOGIA

O desenvolvimento do trabalho foi dividido em 3 etapas. A primeira etapa consistiu no desenvolvimento do software responsável por fazer a interação com a rede interna automotiva. A segunda etapa foi caracterizada pela configuração do *Raspberry Pi* e instalação do software no dispositivo e a terceira etapa se refere à integração da aplicação com um serviço de computação em nuvem.

5.1 DESENVOLVIMENTO DO SOFTWARE DE LEITURA

O objetivo principal da concepção do software nesta fase inicial é possibilitar ao computador interagir com a rede interna veicular, enviando e recebendo dados através da interface *OBD-II*. Para permitir a tradução de protocolos e tornar a comunicação simplificada, foi adquirido o dispositivo ELM327 com transmissão *Bluetooth*, responsável por tratar das conversões dos protocolos automotivos presentes no conector *OBD-II* para uma interface serial padrão, estabelecendo uma comunicação com o adaptador *Bluetooth* do computador.

A construção do software foi feita utilizando-se a linguagem de programação Java na versão 8, juntamente com as seguintes tecnologias: *BlueCove*, *obd-java-api* e *JavaFx*. *BlueCove* é uma biblioteca Java que segue a especificação JSR-82, permitindo que uma aplicação acesse o adaptador *Bluetooth* local para se comunicar com outros dispositivos (BlueCove Team, 2008a). *obd-java-api*, encontrada no repositório de Pires (<https://github.com/pires/obd-java-api>), é uma biblioteca que facilita a comunicação com o adaptador ELM327. O *framework JavaFx* fornece um conjunto de pacotes que facilitam a criação de uma interface gráfica para uma aplicação (Pawlan, 2013). A Figura 16 representa o protótipo do software desenvolvido utilizando o *JavaFx*.

Figura 16 – Imagem da tela de leitura prototipada.



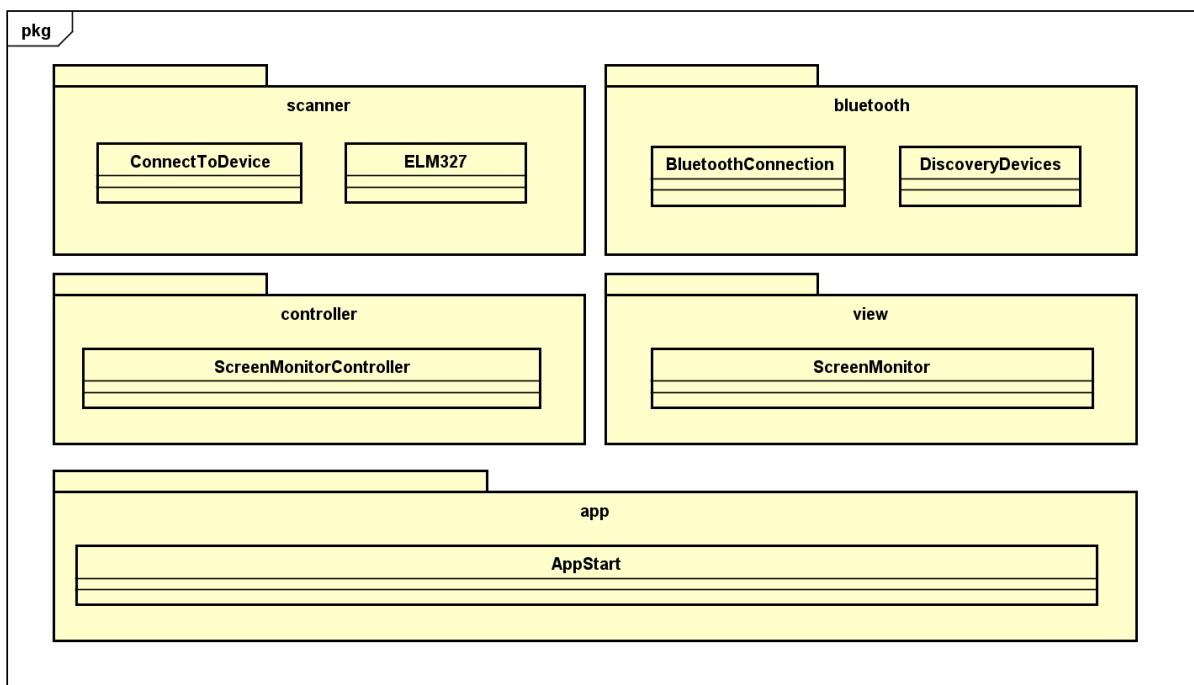
Fonte: produzido pelo autor

5.1.1 ARQUITETURA DO SOFTWARE

O projeto foi desenvolvido utilizando-se o paradigma de programação orientada à objeto e o padrão *Model-View-Controller (MVC)*, e devido a isso, foi necessário criar cinco pacotes para organizar as classes do código fonte. Os nomes dos pacotes criados foram: *app*, *bluetooth*, *controller*, *scanner*, e *view*.

No pacote *app* contém a classe *AppStart* que é responsável por iniciar a aplicação e carregar a interface de usuário *ScreenMonitor* contida no pacote *view*. O pacote *bluetooth* contém duas classes, uma chamada *DiscoveryDevices* – responsável pela descoberta de dispositivos *bluetooth* próximos – e outra chamada *BluetoothConnection*, responsável por obter uma conexão *bluetooth*. Dentro do pacote *scanner* existem duas classes, uma com o nome *ConnectToDevice*, que é responsável por estabelecer uma conexão com o dispositivo ELM327, e outra com o nome *ELM327*, representando o próprio dispositivo. O pacote *view* contém o arquivo *ScreenMonitor* no formato FXML, responsável por representar a interface de usuário. Por fim, o pacote *controller* contém a classe *ScreenMonitorController*, responsável por receber as entradas do usuário na interface e mapear as ações a serem tomadas pelo software. Na Figura 17, é possível observar a organização e estruturação das classes no projeto. A implementação do código das classes dos pacotes *bluetooth* e *scanner* poderá ser consultado em detalhes no capítulo ?? Anexo A.

Figura 17 – Diagrama de pacote exibindo a estrutura de pacotes do projeto com as respectivas classes.



Fonte: produzido pelo autor

No pacote *scanner* existe a classe *ELM327* (Figura 18) responsável por estabelecer a comunicação com o dispositivo de mesmo nome. Essa classe recebe como parâmetro em seu construtor dois objetos do tipo *InputStream* e *OutputStream*. Ela também contém alguns métodos,

que são: *disconnect* (Figura 19), responsável por fechar a conexão dos objetos *InputStream* e *OutputStream*; *readRpm* (Figura 20), responsável por efetuar a leitura da rotação por minuto (RPM) do motor, devolvendo uma *string* no formato correto; *readSpeed* (Figura 21), responsável por efetuar a leitura da velocidade atual do automóvel, também retornando uma *string* no formato Km/h; *readFuelPressure* (Figura 22), responsável por obter a pressão do combustível em uma *string*, no formato Psi ou Kilopascal (kPa); *readOilTemp* (Figura 23), responsável por ler a temperatura do óleo do motor e devolver uma *string* com o valor em graus *Celsius* (°C); *readFindFuelType* (Figura 24), responsável por encontrar qual tipo de combustível está sendo utilizado no tanque; *readFuelLevel* (Figura 25), responsável por obter a informação referente ao nível de combustível presente no tanque, e por último, o método *clearBuffer* (Figura 26), que implementa vários comandos que reiniciam a conexão *OBD*, limpam o eco e o cabeçalho, possibilitando apagar o *buffer* presente no dispositivo ELM327 para realizar novas leituras. Todos os métodos pertencentes à esta classe, exceto o *disconnect*, fazem uso de classes e métodos que pertencem à biblioteca obd-java-api. Desta forma, as classes e métodos contidos nesta biblioteca abstraem a implementação de comandos ‘AT’ e ‘*OBD*’, tornando a sua utilização simplificada, bastando apenas chamar o recurso a ser utilizado sem se preocupar com a forma de implementação da funcionalidade.

Figura 18 – Foto da classe ELM327.

```

19
20 public class ELM327 {
21     private OutputStream outStream;
22     private InputStream inStream;
23
24     public ELM327(InputStream inStream, OutputStream outStream) {
25         this.outStream = outStream;
26         this.inStream = inStream;
27     }
28

```

Fonte: produzido pelo autor

Figura 19 – Foto do método *disconnect* da classe ELM327.

```

29     public void disconnect() throws IOException {
30         this.inStream.close();
31         this.outStream.close();
32     }

```

Fonte: produzido pelo autor

Figura 20 – Foto do método *readRpm* da classe ELM327.

```

34     public String readRpm() throws IOException, InterruptedException {
35         RPMCommand commands = new RPMCommand();
36         commands.run(inStream, outStream);
37         return commands.getFormattedResult();
38     }

```

Fonte: produzido pelo autor

Figura 21 – Foto do método *readSpeed* da classe ELM327.

```

39
40  public String readSpeed() throws IOException, InterruptedException {
41      SpeedCommand commands = new SpeedCommand();
42      commands.run(inStream, outStream);
43      return commands.getFormattedResult();
44  }

```

Fonte: produzido pelo autor

Figura 22 – Foto do método *readFuelPressure* da classe ELM327.

```

45
46  public String readFuelPressure() throws IOException, InterruptedException {
47      this.clearBuffer();
48      FuelPressureCommand commands = new FuelPressureCommand();
49      commands.run(inStream, outStream);
50      return commands.getFormattedResult();
51  }
52

```

Fonte: produzido pelo autor

Figura 23 – Foto do método *readOilTemp* da classe ELM327.

```

54
55  public String readOilTemp() throws IOException, InterruptedException {
56      this.clearBuffer();
57      OilTempCommand commands = new OilTempCommand();
58      commands.run(inStream, outStream);
59      return commands.getFormattedResult();
60  }
61

```

Fonte: produzido pelo autor

Figura 24 – Foto do método *readFindFuelType* da classe ELM327.

```

61
62  public String readFindFuelType() throws IOException, InterruptedException {
63      this.clearBuffer();
64      FindFuelTypeCommand commands = new FindFuelTypeCommand();
65      commands.run(inStream, outStream);
66      return commands.getFormattedResult();
67  }
68

```

Fonte: produzido pelo autor

Figura 25 – Foto do método *readFuelLevel* da classe ELM327.

```

68
69  public String readFuelLevel() throws IOException, InterruptedException {
70      this.clearBuffer();
71      FuelLevelCommand commands = new FuelLevelCommand();
72      commands.run(inStream, outStream);
73      return commands.getFormattedResult();
74  }

```

Fonte: produzido pelo autor

Os pacotes *view*, e *controller* são implementações da arquitetura *MVC*, que correspondem respectivamente à arquivos relacionados à interface de usuário, e à classes responsáveis por

Figura 26 – Foto do método *clearBuffer* da classe ELM327.

```

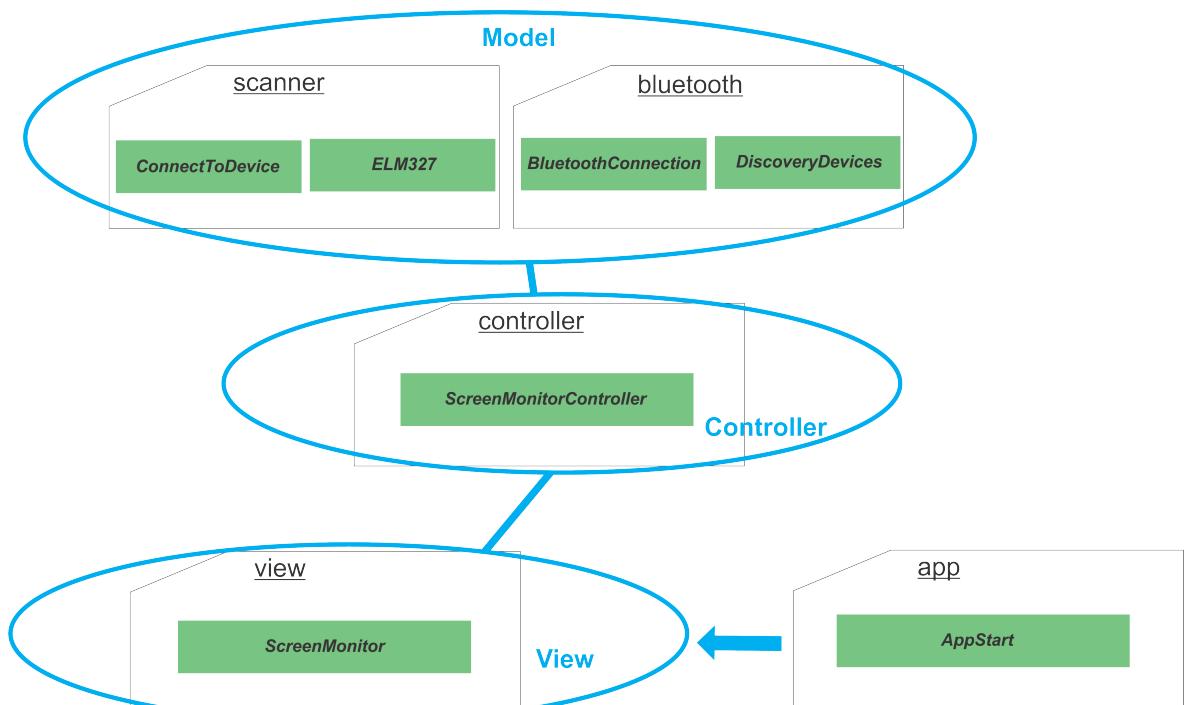
76    private void clearBuffer() throws IOException, InterruptedException {
77        new ObdResetCommand().run(inStream, outStream);
78        new EchoOffCommand().run(inStream, outStream);
79        new HeadersOffCommand().run(inStream, outStream);
80        new SelectProtocolCommand(ObdProtocols.AUTO).run(inStream, outStream);
81    }

```

Fonte: produzido pelo autor

administrar as entradas dos usuários. Segundo Medeiros (2013), a arquitetura *MVC* possui o controlador (*Controller*) que gerencia as entradas dos usuários através das visões (*Views*), passando os comandos para os modelos (*Models*) que gerencia diversos elementos de dados. Seguindo esta ideia, os pacotes que tem o papel de modelo, segundo esta arquitetura, seria o *scanner* e o *bluetooth* (Figura 27).

Figura 27 – Representação da arquitetura *MVC* do projeto.



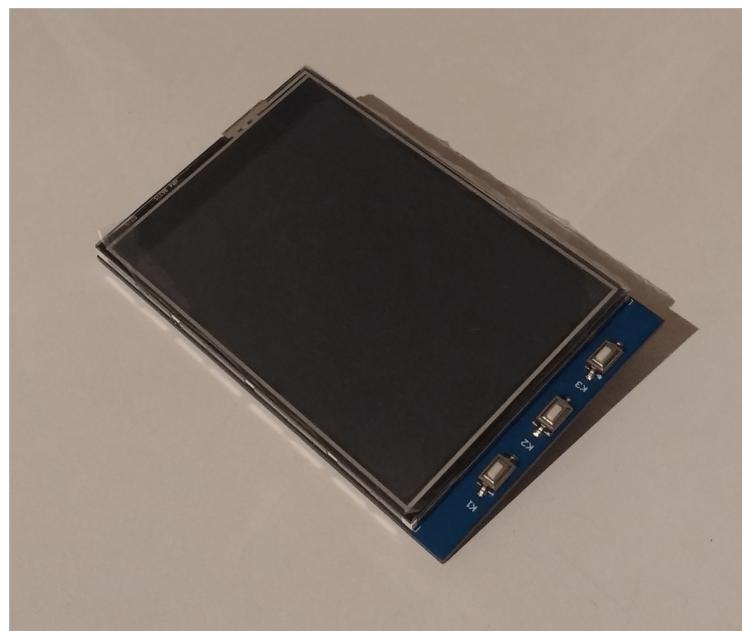
Fonte: produzido pelo autor

5.2 CONFIGURAÇÃO DO RASPBERRY PI E INSTALAÇÃO DO SOFTWARE

Primeiramente foi instalado a versão *Jessie* do *Raspbian*, com a atualização de 05 de julho de 2017 disponível em <https://downloads.raspberrypi.org/raspbian/images/raspbian-2017-07-05/>. O *Raspbian*, conforme menciona Long (2015), é uma distribuição do Linux baseada no Debian. Foi adquirido também, para o *Raspberry* o display de 3.2 polegadas, modelo 3.2inch RPi Display com resolução de 320x240 pixels (Figura 28). Este display é conectado na porta genérica de entrada e saída do *Raspberry Pi* (porta GPIO - Figura 29). O *driver* de instalação

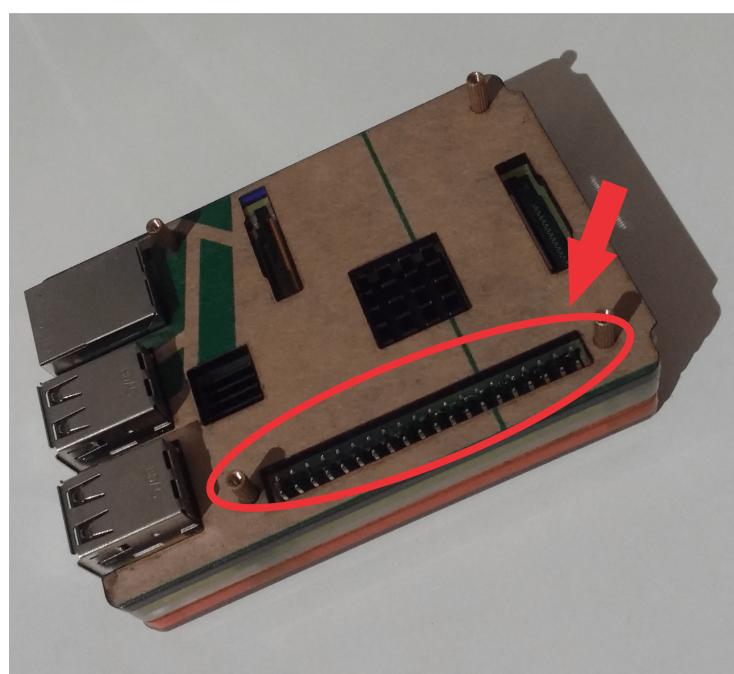
do display está disponível em [https://www.waveshare.com/wiki/3.2inch_RPi_LCD_\(B\)](https://www.waveshare.com/wiki/3.2inch_RPi_LCD_(B)). De acordo com as instruções de instalação presentes neste site, os *drivers* não são compatíveis com sistemas instalados pelo NOOBS. O sistema NOOBS, segundo o Raspberry Pi Foundation, é um instalador de sistema operacional que contém o *Raspbian*.

Figura 28 – Foto do display do *Raspberry Pi*.



Fonte: produzido pelo autor

Figura 29 – Porta GPIO do *Raspberry Pi*.



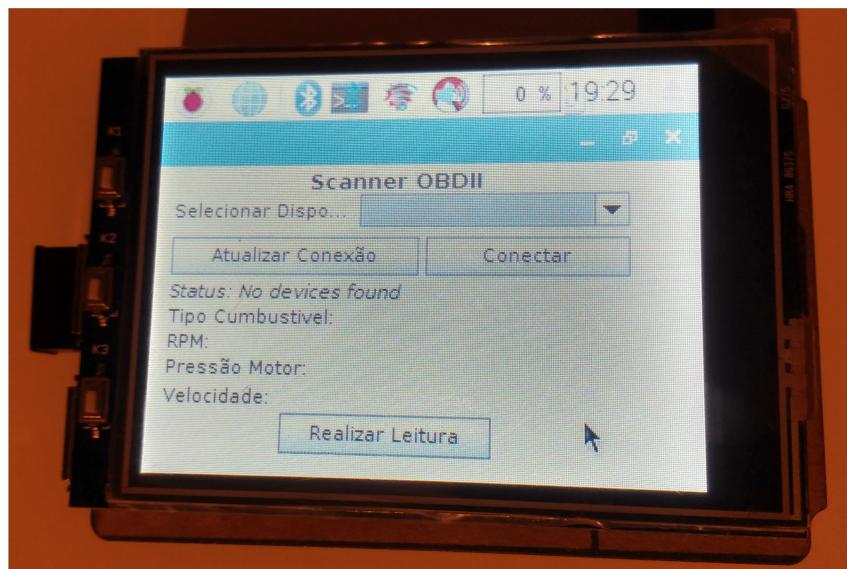
Fonte: produzido pelo autor

Para suportar a instalação e execução do software, depois de instalado e configurado o sistema operacional e o display, foi instalado o kit de desenvolvimento para Java na versão 8

(JDK 8) que incluía a máquina virtual (JRE) para rodar o projeto. Durante a fase de execução do projeto no ambiente do *Raspberry*, notou-se que houve duas incompatibilidades. A primeira incompatibilidade identificada foi relacionada à biblioteca *BlueCove*, pois ela não fornecia suporte para arquitetura ARM. A solução alternativa para contornar este problema foi encontrar uma versão da biblioteca construída para esta arquitetura. A segunda incompatibilidade foi relacionada ao *framework JavaFx*, utilizado na interface gráfica. Foi identificado que nas configurações do *Raspberry* não era possível executar o software utilizando esta tecnologia. Para solucionar este problema, foi necessário redesenhar a interface utilizando o *Swing* – conjunto de componentes que permite a criação de interface gráfica –, que são inteiramente implementados na linguagem Java (Oracle, 2017).

Foram feitas algumas modificações no projeto e na interface para se adequar à tela do *Raspberry* de 3.2 polegadas. Como havia uma limitação de espaço de exibição no display, apenas as leituras de RPM, velocidade, tipo e pressão do combustível foram implementadas. A Figura 30 mostra o software adaptado à tela, sendo executado no *Raspberry Pi*.

Figura 30 – Software sendo executado no *Raspberry Pi*.



Fonte: produzido pelo autor

5.3 INTEGRAÇÃO DA APLICAÇÃO COM SERVIÇO DE COMPUTAÇÃO EM NUVEM

Primeiramente foi feita uma análise da arquitetura que o servidor deveria possuir além de levantar quais serviços e software seriam necessários para garantir o funcionamento do sistema na web. Baseando-se nestas informações, foi adquirido um servidor do tipo *Elastic Compute Cloud (EC2)* da Amazon, pertencente à categoria t2.micro, contendo 1Gb de memória RAM, processador Intel Xeon de 2.5GHz e 8Gb de armazenamento do tipo *Elastic Block Store (EBS)*, rodando o sistema operacional Ubuntu Server 16.04 LTS. Esta categoria de servidor

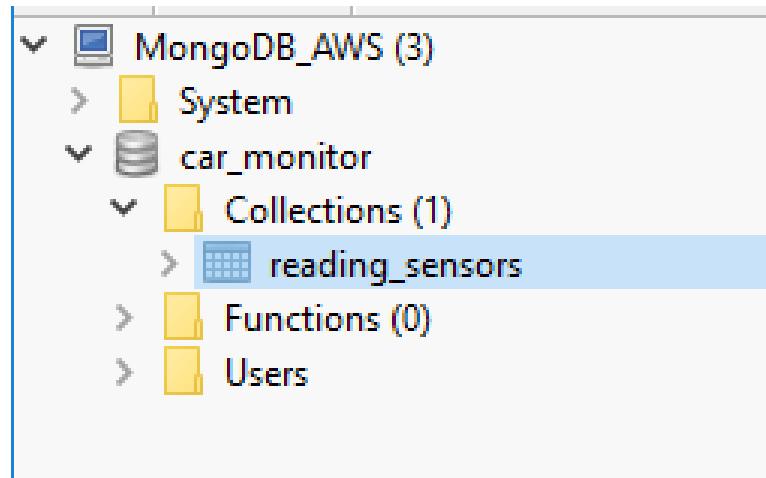
permite o uso por um ano de forma gratuita. O *EC2*, segundo a Amazon (2017a), é um *web service* que disponibiliza capacidade computacional segura e redimensionável na nuvem. O *EBS* disponibiliza volumes de armazenamento para uso com instâncias de servidores do tipo *EC2*.

5.3.1 INSTALAÇÃO E CONFIGURAÇÃO DA BASE DE DADOS NA NUVEM

Depois de feita a aquisição do servidor, a próxima etapa foi instalar e configurar um serviço de banco de dados. Para esta situação foi escolhido o MongoDB, que é um banco de dados não relacional baseado em documentos *JSON* (MongoDB, 2017b). Este banco possui um esquema de dados flexível, permitindo o mapeamento de um documento para uma entidade ou objeto. Cada documento é armazenado dentro de uma coleção do banco. De acordo com o manual do MongoDB (2008), os modelos de dados desnormalizados permitem que as aplicações alterem a estrutura dos documentos contidos nas coleções durante o armazenamento de dados realizando apenas uma única operação no banco de dados sem se preocupar com a sua remodelagem.

Logo após a instalação do banco, foi criado uma base de dados no MongoDB com o nome *car_monitor*, e dentro desta base, foi criado uma coleção com o nome *reading_sensors*. Esta coleção armazenará todos os dados do veículo que foram lidos pelo *Raspberry Pi*. A Figura 31 representa a estrutura do banco de dados que será responsável por armazenar as informações.

Figura 31 – Foto da estrutura do banco de dados.



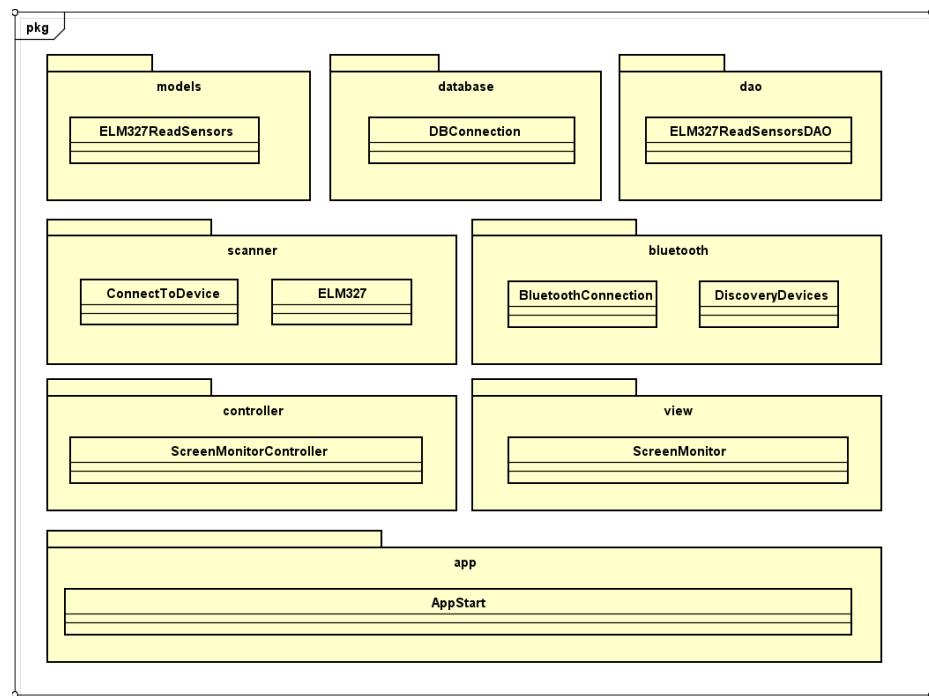
Fonte: produzido pelo autor

5.3.2 INTEGRAÇÃO DO SOFTWARE COM A BASE DE DADOS

Para a integração do software com a base de dados foi necessário instalar as seguintes dependências do MongoDB no projeto: *mongodb-driver*, *mongodb-driver-core* e *bson*, todas na versão 3.5. Para manipular as informações no banco de dados pelo software, foi decidido seguir o modelo *Data Access Object (DAO)*. Segundo a definição de Deepak, Dan & John (apud Medeiros, 2016), o padrão *DAO* abstrai e encapsula todos os acessos ao banco de dados, gerenciando a

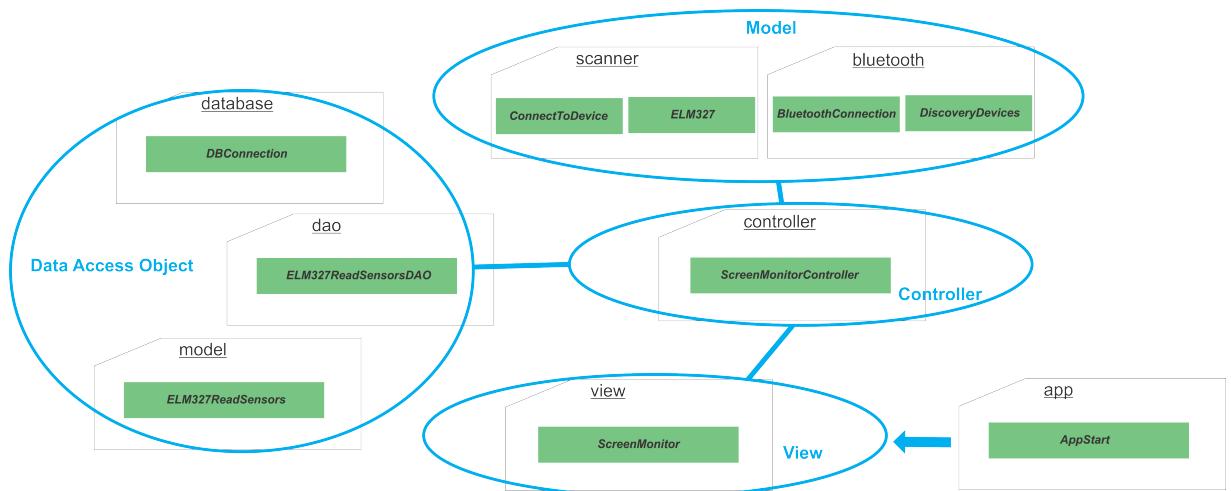
conexão com a base para obter e armazenar as informações. Para implementar este padrão, foi necessário alterar a estrutura do projeto, com a criação de 3 pacotes adicionais: o pacote *dao*, o pacote *database* e o pacote *models*. A Figura 32 mostra a nova estrutura de pacotes e a Figura 33 exibe a representação dos padrões utilizados no projeto baseado nos pacotes implementados.

Figura 32 – Diagrama de pacote exibindo a nova estrutura de pacotes do projeto com as respectivas classes.



Fonte: produzido pelo autor

Figura 33 – Representação da arquitetura *MVC* com o padrão *DAO* do projeto.



Fonte: produzido pelo autor

O pacote *model* contém a classe modelo *ELM327ReadSensors* (Figura 34), que será utilizada para a criação de objetos que armazenarão todas as informações que foram lidas do

veículo. Esta classe possui os campos do tipo *string* referentes à estas informações, que são: modeloCarro, chassiCarro, rpm, velocidade, pressaoCombustivel e tipoCombustivel.

Figura 34 – Foto da classe *ELM327ReadSensors*.

```

2
3 public class ELM327ReadSensors {
4     private String modeloCarro, chassiCarro, rpm,
5             velocidade, pressaoCombustivel, tipoCombustivel;
6
7     public ELM327ReadSensors() {
8
9     }
10
11     public ELM327ReadSensors(String chassiCarro, String modeloCarro) {
12         this.modeloCarro = modeloCarro;
13         this.chassiCarro = chassiCarro;
14     }
15
16     public ELM327ReadSensors(String modeloCarro, String chassiCarro,
17             String rpm, String velocidade, String pressaoCombustivel,
18             String tipoCombustivel) {
19         this.modeloCarro = modeloCarro;
20         this.chassiCarro = chassiCarro;
21         this.rpm = rpm;
22         this.velocidade = velocidade;
23         this.pressaoCombustivel = pressaoCombustivel;
24         this.tipoCombustivel = tipoCombustivel;
25     }

```

Fonte: produzido pelo autor

Analizando o pacote *database*, ele traz a classe *DBConnection* (Figura 35) que é responsável por criar a conexão com o banco de dados na nuvem. Esta classe contém o método estático *getConnection* que retorna um objeto do tipo *MongoDatabase*, pertencente à biblioteca *mongodb-driver*.

Figura 35 – Foto da classe *DBConnection*.

```

2
3 import com.mongodb.MongoClient;[]
5
6 public class DBConnection {
7     public static MongoDatabase getConnection() {
8         MongoClient connect = new MongoClient("18.231.62.135", 27017);
9         return connect.getDatabase("car_monitor");
10    }
11 }
12

```

Fonte: produzido pelo autor

No pacote *dao*, existe a classe *ELM327ReadSensorsDAO* (Figura 36), que implementa o padrão *DAO*. Esta classe abstrai o acesso ao banco de dados, fornecendo um método para inserção chamada *insertDB*, que recebe como parâmetro um objeto do tipo *ELM327ReadSensors*. Este método é responsável por receber este objeto e estruturá-lo em um outro objeto no formato *JSON* para ser inserido no banco de dados. Os dados contidos no MongoDB são documentos *JSON* codificados, conhecidos como *BSON* (MongoDB, 2017a).

Figura 36 – Foto da classe *ELM327ReadSensorsDAO*.

```

2
3  import java.time.LocalDate;
4
5  public class ELM327ReadSensorsDAO {
6      private MongoCollection<Document> collection;
7
8      public ELM327ReadSensorsDAO() {
9          MongoDatabase database = DBConnection.getConnection();
10         collection = database.getCollection("reading_sensors");
11     }
12
13     public void insertDB(ELM327ReadSensors readSensors) {
14         Document bson = new Document();
15
16         bson.append("chassi", readSensors.getChassiCarro())
17             .append("modelo", readSensors.getModeloCarro())
18             .append("rpm", readSensors.getRpm())
19             .append("pressao_combustivel", readSensors.getPressaoCombustivel())
20             .append("tipo_combustivel", readSensors.getTipoCombustivel())
21             .append("velocidade", readSensors.getVelocidade())
22             .append("data", LocalDate.now().toString())
23             .append("hora", LocalTime.now().toString());
24
25         collection.insertOne(bson);
26     }
27
28 }
29
30
31
32
33
34
35
36
37

```

Fonte: produzido pelo autor

Para permitir o upload dos dados na nuvem utilizando o *Raspberry Pi*, foi levantado a necessidade de utilizar uma rede móvel 4G. Desta forma, seria possível conectar o software com o MongoDB localizado em uma instância da Amazon na nuvem.

5.3.3 INSTALAÇÃO E CRIAÇÃO DO WEB SERVICE

Depois de integrado o software com o banco de dados, foi necessário criar um *web service* que seria responsável por realizar as consultas no banco e disponibilizá-las para alguma outra aplicação poder consumir estas informações. Baseado nesta definição, foi escolhido a plataforma Node.js na versão 6.11 para a criação do *web service* utilizando a linguagem *JavaScript*. Segundo o W3Schools (2017), o Node.js é uma plataforma para servidor de código aberto que utiliza a linguagem *JavaScript* para as aplicações de *web service*, permitindo a manipulação de informações no *backend*.

Foi necessário a instalação de três pacotes do Node.js para a criação do *web service*: o pacote *express*, responsável pela abstração das rotas, o pacote *body-parser* responsável por efetuar as conversões *JSON* e o pacote *mongoose*, responsável por mapear os objetos do MongoDB. O *web service* foi implementado apenas para requisições HTTP do tipo *GET* na rota '*/collection*'. Desta maneira, qualquer aplicação que fazer uma solicitação do tipo *GET* para esta rota, será chamado uma função no *web service* (Figura 37) que é responsável por realizar a consulta no banco de dados e retornar os valores encontrados em formato de uma lista de objetos *JSON* respondendo à requisição.

Figura 37 – Foto do método js responsável por responder a solicitações *GET* na rota '/collection'.

```

50
51   app.get('/collection', function (request, response) {
52     collection.find({}, function (error, data) {
53       if(error){
54         response.json({error: 'Não foi possível retornar os dados'});
55       }else{
56         response.json(data);
57       }
58     });
59   });
60

```

Fonte: produzido pelo autor

5.3.4 INSTALAÇÃO E CONFIGURAÇÃO DO SERVIDOR WEB E CRIAÇÃO DA PÁGINA

Após concluída a etapa anterior, seria preciso instalar um servidor web para hospedar a página que iria consumir os dados disponibilizados pelo *web service*. Foi escolhido o servidor Apache para esta finalidade. As configurações padrão do servidor Apache foram mantidas, pois não haveria necessidade de alterá-las nesta situação.

O desenvolvimento da página web foi feito utilizando-se as tecnologias HTML, *JQuery* e *Bootstrap*. A função desta página é simplesmente consumir as informações disponíveis no *web service*, não possuindo nenhuma outra funcionalidade específica. A página contém um componente de seleção, que permite escolher o tipo de filtro para exibir os resultados. Logo abaixo contém o campo de entrada que permite inserir o valor da informação que se deseja filtrar. A Figura 38 apresenta os componentes de filtro mencionados acima.

Figura 38 – Foto dos elementos de filtro da página.

Fonte: produzido pelo autor

Após preenchido o valor, existe um botão chamado monitorar, que ao acionado, faz a requisição *GET* via *AJAX* (Figura 39) para o *web service* e retorna os valores na tabela de acordo com o filtro informado. Por fim, a Figura 40 mostra toda a integração do sistema com a nuvem, representando as trocas de dados e comunicação entre diferentes serviços.

Figura 39 – Foto do método *JQuery* responsável pelo *AJAX*.

```

19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42

```

```

$('#btn-monitorar').click(function(event){
    event.preventDefault();

    var input_filtro = $('#input-filtro').val();

    $('#corpo-monitoramento tr').remove();
    $.get('http://18.231.62.135:5000/collection',function(dataReceived){

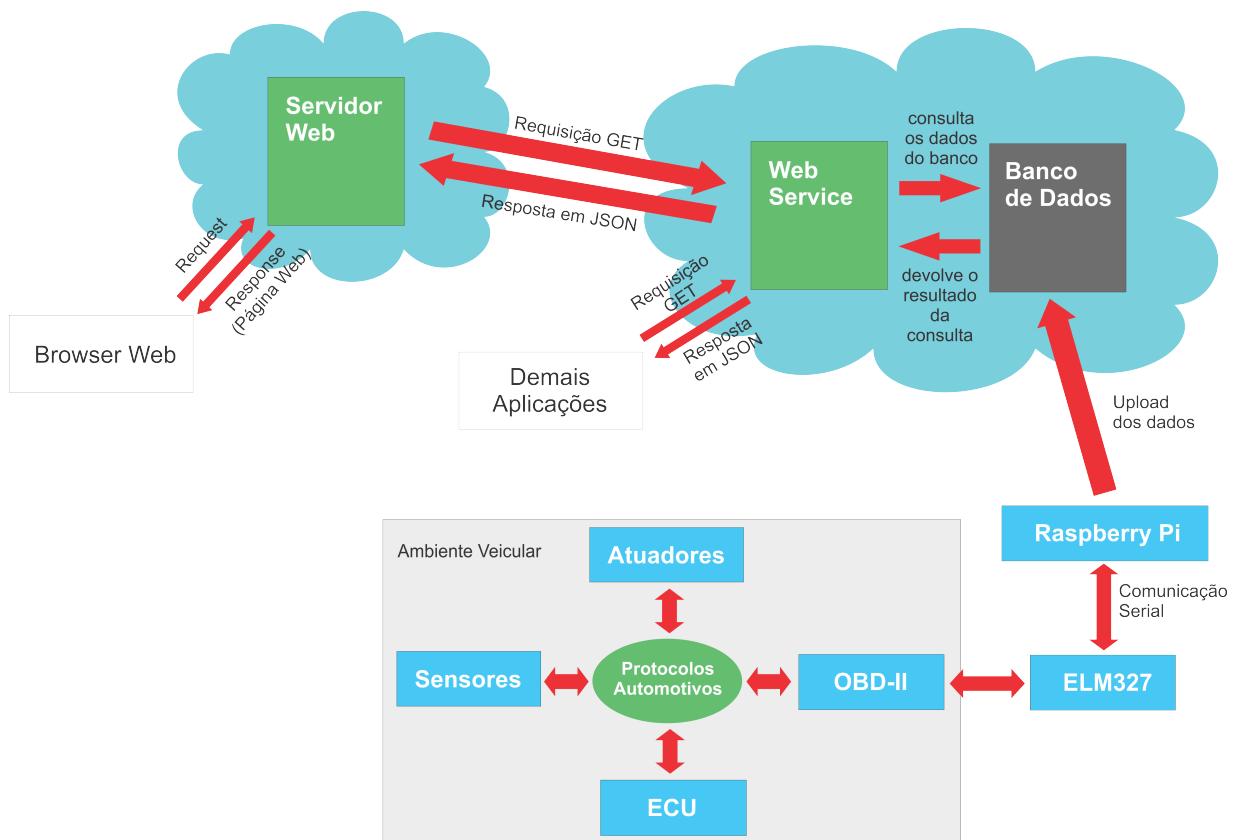
        $(dataReceived).each(function(){

            if(filtro_select == 'chassi' && input_filtro == this.chassi){
                tbody.append(criaTr(this.data, this.hora, this.rpm, this.velocidade,
                    this.tipo_combustivel, this.pressao_combustivel, this.modelo));
            }
            if(filtro_select == 'modelo' && input_filtro == this.modelo){
                tbody.append(criaTr(this.data, this.hora, this.rpm, this.velocidade,
                    this.tipo_combustivel, this.pressao_combustivel, this.chassi));
            }
        });
    });
});

```

Fonte: produzido pelo autor

Figura 40 – Diagrama representando a comunicação do sistema com a nuvem.



Fonte: produzido pelo autor

6 TESTES E AVALIAÇÃO DOS RESULTADOS

Durante o desenvolvimento do sistema como um todo, desde a sua fase local de desenvolvimento do software até a integração com os serviços da computação em nuvem, foram feitos diversos testes para garantir que as funcionalidades básicas estipuladas no início estavam sendo atendidas. Desta forma, será abordado nesta seção como foram realizadas cada etapa dos testes, e os principais resultados obtidos.

6.1 TESTE DO SOFTWARE DE LEITURA

Depois de desenvolvido o software, foram realizados os testes de funcionalidade no ambiente de desenvolvimento (notebook) para analisar se o funcionamento estava conforme o esperado. Nesta etapa, verificou-se que algumas leituras obtidas do ELM327 estavam retornando valores que não condiziam com a realidade. Contudo, estes valores eram tratados pela biblioteca obd-java-api, disparando uma *exception* da própria biblioteca referente ao valor retornado. Baseado no resultado deste teste, três hipóteses que levantadas: na primeira, possivelmente o PID referente à leitura não era suportado (ou não foi implementado) pelo veículo ou protocolo utilizado; na segunda, possivelmente por questões de segurança, o próprio sistema do veículo negou a requisição para o PID; ou ainda, poderia haver a possibilidade do dispositivo estar retornando algum dado inválido, podendo ser lixo de memória.

Depois de feita a análise do problema, foram realizados mais testes direcionados à exploração desta falha visando uma possível correção. Foi necessário também consultar a documentação do dispositivo ELM327 para entender como ele trabalhava com as solicitações. Foi descoberto que o dispositivo era capaz de retransmitir a leitura anterior, semelhante a um eco. Isso é possível pois o dispositivo contém uma pequena memória de *buffer* que armazena o último comando enviado para o dispositivo. Baseando-se nestas informações coletadas, foi decidido implementar o método *clearBuffer* na classe ELM327, contendo alguns comandos ‘AT’ que eram responsáveis por apagar o eco e reiniciar a comunicação, para garantir que o *buffer* estivesse limpo para a nova leitura.

Após realizar a alteração proposta, foi observado que algumas das leituras que antes retornavam uma *exception* estavam agora retornando valores válidos e formatados segundo a implementação da biblioteca, evidenciando que parte do problema foi solucionado. Entretanto, algumas leituras ainda continuaram a retornar as *exceptions*. Baseando-se nestas observações, para as leituras que foram possíveis após a alteração mencionada, uma explicação aceitável seria a possibilidade do *buffer* estar sujo. Para explicar as outras leituras que persistiram com o problema mencionado, uma das duas primeiras hipóteses poderia ser a verdadeira.

Além da questão levantada acima, observou-se também certa lentidão durante a inici-

alização do software e durante a realização de algumas leituras. Inicialmente foi levantado a hipótese de que a comunicação *bluetooth* que estava causando a lentidão, entretanto, quando foi implementado a mesma lógica pelo console, notou-se que a execução dele foi mais rápido. Baseado neste teste, concluiu-se que a implementação contendo a interface gráfica estava deixando a execução do software mais lenta. Entretanto, apesar desta análise e conclusão, foi mantida a interface do sistema.

6.2 TESTE DE INTEGRAÇÃO DO SOFTWARE COM O RASPBERRY PI

Depois de preparado o ambiente do *Raspberry Pi*, foi executado o software no dispositivo para análise e testes. Conforme já apresentado na metodologia, foram identificadas algumas incompatibilidades, solucionadas logo na sequência. Contudo, depois das alterações, notou-se que o software foi executado mais rápido em comparação com a primeira execução no ambiente de desenvolvimento, mas ainda demorava algum tempo em alguns momentos da execução. Analisando-se o resultado dos testes, foi levantada a hipótese de a lentidão estar sendo causada pelo fato de o software estar rodando no dispositivo através da máquina virtual do Java (*Java Virtual Machine – JVM*), combinado com a renderização da interface.

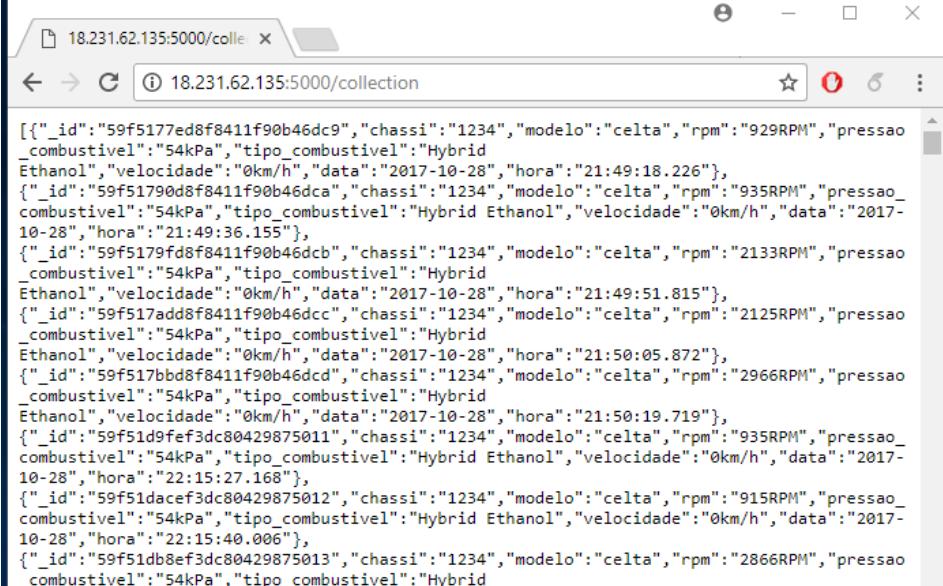
A fim de tornar a execução mais rápida, foi feita uma análise técnica para estudar a viabilidade de migração do software para a linguagem *Python*. Além do *Raspberry* suportar scripts *Python*, Richardson & Wallace (2013) afirmam que esta linguagem tende a ser mais rápida pelo fato de ser interpretada. Durante a análise, foi encontrado uma biblioteca em *Python* que trazia diversas funções que permitiam a comunicação com o ELM327, assim como a biblioteca para Java obd-java-api. Esta biblioteca para *Python* foi encontrada em <http://python-obd.readthedocs.io/en/latest/>, junto com a documentação explicando as principais funções presentes nela. Apesar desta alternativa parecer, em primeiro momento, uma solução viável, houve alguns contratemplos, que levaram a permanência do software na linguagem Java. Uma das dificuldades que tornou inviável a mudança foi o esforço demasiado grande para reestruturar todo o software na nova linguagem, considerando que todas as partes funcionais já estavam prontas em Java.

6.3 TESTES DE INTEGRAÇÃO DA APLICAÇÃO COM A COMPUTAÇÃO EM NUVEM

Após finalizada toda a parte de integração com os recursos da computação em nuvem, desde banco de dados até o *web service*, foram realizados diversos testes para garantir a comunicação entre os sistemas. Nesta fase, não houve dificuldades e todo o sistema se comunicou conforme o esperado. Para garantir a comunicação do software com o banco de dados na nuvem, foi utilizada uma infraestrutura de rede móvel 4G do celular, que roteava o sinal via Wi-Fi para o *Raspberry Pi*, conforme já abordado anteriormente. O teste foi realizado com o veículo em

movimento, e foi notado que durante a execução, a única limitação que foi observada é que ao entrar em uma região sem cobertura 4G, o sistema não fazia upload das informações, o que resultava em perda dos dados que foram coletados. Contudo, toda informação enviada ao banco de dados era disponibilizado pelo *web service* em formato de uma lista de objetos *JSON* (Figura 41) para ser consumido por outra aplicação.

Figura 41 – Foto dos objetos *JSON* disponibilizados pelo *web service*.



The screenshot shows a web browser window with the URL `18.231.62.135:5000/collection`. The page displays a JSON array of vehicle data. Each object in the array represents a vehicle's status at a specific time. The fields include `_id`, `chassi`, `modelo`, `rpm`, `pressao_combustivel`, `tipo_combustivel`, `Ethanol`, `velocidade`, `data`, and `hora`. The data shows various vehicle models (celta), engine speeds (e.g., 929RPM, 935RPM, 2133RPM, 2125RPM, 2966RPM, 915RPM, 2866RPM), and fuel types (Hybrid, Hybrid Ethanol). The timestamp indicates data from October 28, 2017, at various times between 21:49:18 and 22:15:27.

```
[{"_id": "59f5177ed8f8411f90b46dc9", "chassi": "1234", "modelo": "celta", "rpm": "929RPM", "pressao_combustivel": "54kPa", "tipo_combustivel": "Hybrid Ethanol", "velocidade": "0km/h", "data": "2017-10-28", "hora": "21:49:18.226"}, {"_id": "59f51790d8f8411f90b46dca", "chassi": "1234", "modelo": "celta", "rpm": "935RPM", "pressao_combustivel": "54kPa", "tipo_combustivel": "Hybrid Ethanol", "velocidade": "0km/h", "data": "2017-10-28", "hora": "21:49:36.155"}, {"_id": "59f5179fd8f8411f90b46dcbb", "chassi": "1234", "modelo": "celta", "rpm": "2133RPM", "pressao_combustivel": "54kPa", "tipo_combustivel": "Hybrid Ethanol", "velocidade": "0km/h", "data": "2017-10-28", "hora": "21:49:51.815"}, {"_id": "59f517addsf8411f90b46dcc", "chassi": "1234", "modelo": "celta", "rpm": "2125RPM", "pressao_combustivel": "54kPa", "tipo_combustivel": "Hybrid Ethanol", "velocidade": "0km/h", "data": "2017-10-28", "hora": "21:50:05.872"}, {"_id": "59f517bbd8f8411f90b46dcdb", "chassi": "1234", "modelo": "celta", "rpm": "2966RPM", "pressao_combustivel": "54kPa", "tipo_combustivel": "Hybrid Ethanol", "velocidade": "0km/h", "data": "2017-10-28", "hora": "21:50:19.719"}, {"_id": "59f51d9fef3dc80429875011", "chassi": "1234", "modelo": "celta", "rpm": "935RPM", "pressao_combustivel": "54kPa", "tipo_combustivel": "Hybrid Ethanol", "velocidade": "0km/h", "data": "2017-10-28", "hora": "22:15:27.168"}, {"_id": "59f51dacef3dc80429875012", "chassi": "1234", "modelo": "celta", "rpm": "915RPM", "pressao_combustivel": "54kPa", "tipo_combustivel": "Hybrid Ethanol", "velocidade": "0km/h", "data": "2017-10-28", "hora": "22:15:40.006"}, {"_id": "59f51db8ef3dc80429875013", "chassi": "1234", "modelo": "celta", "rpm": "2866RPM", "pressao_combustivel": "54kPa", "tipo_combustivel": "Hybrid Ethanol", "velocidade": "0km/h", "data": "2017-10-28", "hora": "22:15:40.006"}]
```

Fonte: produzido pelo autor

Por fim, foi testado com sucesso o consumo dos dados por meio de uma página web (Figura 42), através de requisições utilizando o *AJAX* do *JQuery* pelo *frontend*.

Figura 42 – Foto da página web estruturando em uma tabela os dados JSON.

Monitoramento veicular

Filtrar resultados por:

Chassi

Informe o chassi do veículo:

1234

Monitorar

Dados do veiculo:

Dados da Medição						
Data	Hora	RPM	Velocidade	Tipo de Combustível	Pressão do Combustível	Modelo
2017-10-28	21:49:18.226	929RPM	0km/h	Hybrid Ethanol	54kPa	celta
2017-10-28	21:49:36.155	935RPM	0km/h	Hybrid Ethanol	54kPa	celta
2017-10-28	21:49:51.815	2133RPM	0km/h	Hybrid Ethanol	54kPa	celta

Fonte: produzido pelo autor

7 CONCLUSÕES

Este trabalho mostrou um estudo detalhado da arquitetura presente nos sistemas embarcados automotivos e também a forma com que estes sistemas se comunicam, apresentando as categorias e protocolos utilizados, e como o sistema é dividido. Neste estudo também foi observada a limitação de informações disponibilizadas pelas montadoras sobre os sistemas eletrônicos presentes em seus automóveis. Foi visto também que é fundamental a realização do diagnóstico para a prevenção de eventuais problemas, e a importância do estudo de aplicações voltadas ao cenário automotivo, buscando integrá-lo à tecnologia da informação. Na sequência, foram explorados também alguns conceitos de informática que puderam se relacionar diretamente e indiretamente com o ambiente de um automóvel e como esses conceitos poderiam se cruzar, permitindo aplicações possivelmente viáveis aplicadas à internet das coisas.

Com base neste estudo, foi proposto o desenvolvimento de um sistema responsável por ler as informações relacionadas ao funcionamento do automóvel presentes em uma rede veicular, processar os dados e enviá-los a um banco de dados localizado na nuvem, e a partir disso, disponibilizar estas informações para o acesso de eventuais aplicações, buscando colocar em prática os conceitos estudados ao longo do trabalho. Entretanto, apesar do êxito em efetuar a leitura dos dados presentes nos automóveis, não foi possível verificar se a conversão das informações lidas do automóvel foi precisa por faltar parâmetros de comparação autênticos.

7.1 PROPOSTA PARA TRABALHOS FUTUROS

As dificuldades enfrentadas ao longo do percurso deixam margem para a exploração de outros trabalhos relacionados, além desta proposta. Existem alguns ajustes que podem ser aperfeiçoados numa extensão deste projeto.

Um dos pontos de melhoria é relacionado à tela do *Raspberry Pi* que foi adquirida. Notou-se que a tela de 3.2 polegadas era muito pequena para a exibição das informações, o que levou à adaptação da interface para exibição dos dados. Isso resultou na redução de informações que seriam exibidas na tela. Uma possível sugestão seria a utilização de uma tela de 5 polegadas ou maior.

Outro aspecto que também pode ser explorado é a análise da precisão dos valores que foram obtidos do veículo através do software. É importante comparar os valores gerados pelo sistema com os gerados através de *scanners* e outros equipamentos de leitura originais das montadoras. É importante também analisar os valores que são considerados padrões pelas montadoras junto com as variações permitidas, comparando-as com os valores obtidos.

Analizando-se os resultados referentes ao desempenho do sistema, seria possível considerar a reimplementação do software utilizando a linguagem *Python* para fins de estudo e análise

de eficiência do algoritmo. Mesmo levantada a hipótese de uma possível otimização com esta linguagem, é necessário colocar em prática para a análise do resultado e confirmação da teoria.

Uma sugestão de melhoria futura do sistema está relacionado com o estudo de políticas de segurança, uma vez que o sistema está totalmente conectado na web, e em sua concepção não foi medido os possíveis riscos e o impacto que poderia gerar. A questão da privacidade dos dados que serão disponibilizados também é um assunto que poderá ser discutido e definido métricas ou ações a serem tomadas para não infringí-la.

Analizando toda a estrutura do trabalho, é possível notar que o sistema poderá gerar grande volume de dados de forma diversificada (indo ao encontro do conceito definido por Chede de *Big Data*) tendo uma estrutura totalmente escalável e conectada à internet. Desta forma, observando a infraestrutura utilizada, como o uso de tecnologia de baixo custo, a computação em nuvem e o uso de um banco de dados não relacional, torna-se possível uma extensão de estudo por outras áreas que podem se integrar a este trabalho, como envolvendo aplicações de internet das coisas e *Big Data*.

REFERÊNCIAS

- AMAZON. **Amazon Web Service. Amazon EC2.** 2017. Disponível em: <https://aws.amazon.com/pt/ec2/?nc2=h_m1>.
- AMAZON. **Amazon Web Service. Amazon Elastic Block Store: Armazenamento em blocos persistente para o Amazon EC2.** 2017. Disponível em: <<https://aws.amazon.com/pt-ebs/>>.
- AMAZON. **Amazon Web Service. O que é a computação em nuvem?** 2017. Disponível em: <<https://aws.amazon.com/pt/what-is-cloud-computing/>>.
- ASHTON, Kevin. **RFID Journal. That 'Internet of Things' Thing.** 2009. Disponível em: <<http://www.rfidjournal.com/articles/view?4986>>.
- BLUECOVE TEAM. **BlueCove Team. BlueCove.** 2008. Disponível em: <<http://www.bluecove.org/>>.
- BLUECOVE TEAM. **BlueCove Team. bluecove 2.1.0 API Documentation.** 2008. Disponível em: <<http://www.bluecove.org/bluecove/apidocs/index.html>>.
- CARRO, Luigi; WAGNER, Flávio Rech. Sistemas computacionais embarcados. **Jornadas de atualização em informática. Campinas: UNICAMP,** 2003.
- CASAGRAS, RFID. The inclusive model for the internet of things report. **EU Project**, n. 216803, p. 16–23, 2011.
- CHARETTE, Robert N. **IEEE Spectrum. This Car Runs on Code.** 2009. Disponível em: <<https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code>>.
- CHEDE, Cesar. **IBM Developer Works. Você realmente sabe o que é Big Data?** 2012. Disponível em: <https://www.ibm.com/developerworks/community/blogs/ctaurion/entry/voce_realmente_sabe_o_que_e_big_data?lang=en>.
- DEEPAK, A.; DAN, M.; JOHN, C. **Core J2EE Patterns: Best Practices and Design Strategies.** 2. ed. [S.l.]: Prentice Hall, 2003.
- DIAS, Renata Rampim de Freitas. **Internet das Coisas sem mistérios: Uma nova inteligência para os negócios.** 1. ed. [S.l.]: São Paulo: Netpress Books, 2016. 127 p.
- ELM ELETRONICS. **ELM327: OBD to RS232 Interpreter.** 1. ed. [S.l.], 2012. 76 p.
- FAGUNDES, Felipe Augusto Vieira; SILVA, Gustavo Luiz da; ASSIS, Marco Aurélio Scomparim. **Sistema de monitoramento automotivo remoto.** 2015. Monografia (Tecnologia em Eletrônica Automotiva), FATEC Santo André (Faculdade de Tecnologia de Santo André), Santo André, Brasil.
- GILL, Hellen. **National Science Foundation. Cyber-Physical Systems (CPS).** Disponível em: <<https://www.nsf.gov/pubs/2008/nsf08611/nsf08611.htm>>.
- LEE, Edward Ashford; SESHIA, Sanjit Arunkumar. **Introduction to Embedded Systems: A cyber-physical systems approach.** 2. ed. [S.l.]: MIT Press, 2017. 585 p.

LONG, Simon. **Raspberry Pi Foundation. Jessie is here.** 2015. Disponível em: <<https://www.raspberrypi.org/blog/raspbian-jessie-is-here/>>.

MARQUES, Marco Antonio. **CAN Automotivo: Sistema de Monitoramento.** 2004. Dissertação (Mestrado em Ciências em Engenharia Elétrica na área de Concentração Automação e Sistemas Elétricos Industriais), UNIFEI (Universidade Federal de Itajubá), Itajubá, Brasil.

MEDEIROS, Higor. **DevMedia. Introdução ao Padrão MVC.** 2013. Disponível em: <<https://www.devmedia.com.br/introducao-ao-padroao-mvc/29308>>.

MEDEIROS, Higor. **DevMedia. Implementando o Data Access Object no Java EE.** 2016. Disponível em: <<https://www.devmedia.com.br/implementando-o-data-access-object-no-java-ee/33339>>.

MICROSOFT. **Microsoft Azure. O que é computação em nuvem?** 2017. Disponível em: <<https://azure.microsoft.com/pt-br/overview/what-is-cloud-computing/>>.

MONGODB. **MongoDB. Data Modeling Introduction.** 2008. Disponível em: <<https://docs.mongodb.com/manual/core/data-modeling-introduction/>>.

MONGODB. **MongoDB. JSON and BSON.** 2017. Disponível em: <<https://www.mongodb.com/json-and-bson>>.

MONGODB. **MongoDB. What is MongoDB?** 2017. Disponível em: <<https://www.mongodb.com/what-is-mongodb>>.

NAVET, Nicolas; SIMONOT-LION, Françoise. **Automotive Embedded Systems Handbook.** [S.I.]: CRC press, 2008. 490 p.

OLIVEIRA, Ricardo Merces de. **Raspberry Pi: Conceito & Prática.** [S.I.]: Rio de Janeiro: Editora Ciência Moderna Ltda., 2013.

ORACLE. **Oracle and/or its affiliates. Swing (Java™ Foundation Classes).** 2017. Disponível em: <<https://docs.oracle.com/javase/7/docs/technotes/guides/swing/index.html>>.

PAWLAN, Monica. **Oracle and/or its affiliates. What Is JavaFX?** 2013. Disponível em: <<https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm>>.

RASPBERRY PI FOUNDATION. **Raspberry Pi Foundation. NOOBS.** Disponível em: <<https://www.raspberrypi.org/downloads/noobs/>>.

RICHARDSON, Matt; WALLACE, Shawn. **Getting Started with Raspberry Pi.** 1. ed. [S.I.]: Sebastopol, Califórnia, EUA: O'Reilly Media, 2013. 180 p.

SMITH, Craig. **The car hacker's handbook: A guide for the penetration tester.** 1. ed. [S.I.]: San Francisco, Califórnia, EUA: No Starch Press, 2016. 304 p.

STAROSKI, Ricardo Artur. **OBD-JRP: Monitoramento veicular com Java e Raspberry Pi.** 2016. Monografia (Bacharel em Ciência da Computação), Universidade Regional de Blumenau, Blumenau, Brasil.

TANENBAUM, Andrew S. **Sistemas Distribuídos: Princípios e paradigmas.** 2. ed. [S.I.]: São Paulo: Pearson Prentice Hall, 2007. 402 p.

TORVALDS, Linus. **Linus Torvalds: Linux succeeded thanks to selfishness and trust.** Entrevista. BBC NEWS, 2012. Entrevista concedida a Leo Kelion (Repórter de Tecnologia). Disponível em: <<http://www.bbc.com/news/technology-18419231>>.

W3SCHOOLS. **W3Schools. Node.js Introduction.** 2017. Disponível em: <https://www.w3schools.com/nodejs/nodejs_intro.asp>.

A Apêndice A

Fazendo uma análise do pacote *bluetooth*, existe a classe *BluetoothConnection* que traz um método estático chamado *getConnectionBluetooth* (Figura 43), esperando como parâmetro uma URL de conexão. Este método que tem a responsabilidade de obter uma conexão *bluetooth* devolvendo um objeto do tipo *StreamConnection*, pertencente à biblioteca *BlueCove*. A Classe *DiscoveryDevices* (Figura 44) implementa a interface *DiscoveryListener*, também pertencente à biblioteca *BlueCove*, que permite a descoberta de dispositivos e serviços. Esta interface fornece quatro métodos para serem implementados: dois para descobrir dispositivos, que são o *deviceDiscovered* (Figura 45) – método que é invocado quando é encontrado um dispositivo durante uma consulta – e o *inquiryCompleted* (Figura 46) – método que é chamado quando uma consulta é concluída, e dois para descobrir serviços, que são o *servicesDiscovered* (Figura 47) – são invocados quando os serviços são encontrados durante uma pesquisa por serviços – e o *serviceSearchCompleted* (Figura 48) – são chamados quando uma pesquisa de serviço foi concluída ou encerrada devido a um erro (BlueCove Team, 2008b). Além destes métodos contidos na assinatura da interface, a classe *DiscoveryDevices* também traz o método *discovery* (Figura 49), responsável por iniciar a descoberta de dispositivos.

Figura 43 – Foto do método *getConnectionBluetooth* da classe *BluetoothConnection*.

```

2
3④ import java.io.IOException;
4
5 public class BluetoothConnection {
6     private static StreamConnection stConnection;
7
8     public static StreamConnection getConnectionBluetooth(String connectionUrl)
9         throws IOException {
10        stConnection = (StreamConnection)Connector.open(connectionUrl);
11        return stConnection;
12    }
13}
14
15}
16}
17

```

Fonte: produzido pelo autor

Observando o pacote *scanner*, existe a classe *ConnectToDevice* (Figura 50) que espera como parâmetro em seu construtor um objeto do tipo *DiscoveryDevices*. Esta classe também possui o método *connectToDevice* (Figura 51), que espera como parâmetro um índice referente ao dispositivo que se deseja conectar.

Figura 44 – Foto da Classe *DiscoveryDevices* implementando a interface *DiscoveryListener*.

```
15  
16 public class DiscoveryDevices implements DiscoveryListener{  
17     private Object lock;  
18     private Vector remoteDevices;  
19     private List<String> namesRemoteDevices;  
20     private String connectionUrl;  
21     private LocalDevice localDevice;  
22     private DiscoveryAgent discoveryAgent;  
23     private String status;  
24  
25     public DiscoveryDevices() {  
26         this.lock = new Object();  
27         this.remoteDevices = new Vector();  
28         this.namesRemoteDevices = new ArrayList<String>();  
29     }  
30 }
```

Fonte: produzido pelo autor

Figura 45 – Foto do método *deviceDiscovered* da classe *DiscoveryDevices*.

```
68  
69     @Override  
70     public void deviceDiscovered(RemoteDevice device, DeviceClass dClass) {  
71         if( !remoteDevices.contains(device) )  
72         {  
73             remoteDevices.addElement(device);  
74         }  
75     }
```

Fonte: produzido pelo autor

Figura 46 – Foto do método *inquiryCompleted* da classe *DiscoveryDevices*.

```

75
76@Override
77public void inquiryCompleted(int discType) {
78    synchronized(lock)
79    {
80        lock.notify();
81    }
82    switch(discType)
83    {
84        case DiscoveryListener.INQUIRY_COMPLETED:
85            status = "Status: Inquiry Completed";
86            System.out.println(status);
87            break;
88
89        case DiscoveryListener.INQUIRY_TERMINATED:
90            status = "Status: Inquiry Terminated";
91            System.out.println(status);
92            break;
93
94        case DiscoveryListener.INQUIRY_ERROR:
95            status = "Status: Inquiry Error";
96            System.out.println(status);
97            break;
98
99        default:
100            status = "Status: Unknown Response Code";
101            System.out.println(status);
102        }
103    }
104

```

Fonte: produzido pelo autor

Figura 47 – Foto do método *servicesDiscovered* da classe *DiscoveryDevices*.

```

112
113@Override
114public void servicesDiscovered(int transID, ServiceRecord[] serviceRecord) {
115    if( !(serviceRecord==null) && serviceRecord.length>0 )
116    {
117        connectionUrl=serviceRecord[0].getConnectionURL(0, false);
118    }
119}
120

```

Fonte: produzido pelo autor

Figura 48 – Foto do método *serviceSearchCompleted* da classe *DiscoveryDevices*.

```

104
105@Override
106public void serviceSearchCompleted(int arg0, int arg1) {
107    synchronized(lock)
108    {
109        lock.notify();
110    }
111}
112

```

Fonte: produzido pelo autor

Figura 49 – Foto do método *discovery* da classe *DiscoveryDevices*.

```

30
31  public void discovery() throws BluetoothStateException, IOException {
32      localDevice = LocalDevice.getLocalDevice();
33      discoveryAgent = localDevice.getDiscoveryAgent();
34
35      discoveryAgent.startInquiry(DiscoveryAgent.GIAC, this);
36      try
37      {
38          synchronized (lock)
39          {
40              lock.wait();
41          }
42      } catch(InterruptedException e)
43      {
44          e.printStackTrace();
45      }
46
47      if(remoteDevices.size()<=0)
48      {
49          status = "Status: No devices found";
50          System.out.println(status);
51      }
52      else
53      {
54          namesRemoteDevices.clear();
55          for(int i = 0; i < remoteDevices.size(); i++)
56          {
57              RemoteDevice remote_device = (RemoteDevice)remoteDevices.elementAt(i);
58              System.out.println((i+1)+"."+remote_device.getFriendlyName(true)+" "
59                               +remote_device.getBluetoothAddress());
60              namesRemoteDevices.add((i+1)+"."+remote_device.getFriendlyName(true)+" "
61                               +remote_device.getBluetoothAddress());
62          }
63      }
64
65  }
66
67 }

```

Fonte: produzido pelo autor

Figura 50 – Foto da classe *ConnectToDevice*.

```

15
16  public class ConnectToDevice {
17      private DiscoveryDevices discoveryDevices;
18      private DiscoveryAgent discoveryAgent;
19      private Vector remoteDevices;
20      private Object lock;
21      private String connectionUrl;
22      private String status;
23
24  public ConnectToDevice(DiscoveryDevices discoveryDevices) {
25      this.discoveryDevices = discoveryDevices;
26      this.discoveryAgent = discoveryDevices.getDiscoveryAgent();
27      this.remoteDevices = discoveryDevices.getRemoteDevices();
28      this.lock = discoveryDevices.getLock();
29      this.connectionUrl = discoveryDevices.getConnectionUrl();
30  }
31

```

Fonte: produzido pelo autor

Figura 51 – Foto do método *connectToDevice* da classe *ConnectToDevice*.

```
31
32     public ELM327 connectToDevice(int index) throws IOException {
33         ELM327 scanner = null;
34         RemoteDevice des_device = (RemoteDevice)remoteDevices.elementAt(index);
35         UUID[] uuidset=new UUID[1];
36         uuidset[0]=new UUID("1101",true);
37
38         discoveryAgent.searchServices(null, uuidset, des_device, discoveryDevices);
39         try
40         {
41             synchronized(lock)
42             {
43                 lock.wait();
44             }
45         }
46         catch(InterruptedException e)
47         {
48             e.printStackTrace();
49         }
50
51         if(connectionUrl == null)
52         {
53             status = "Status: Device does not support SPP.";
54             System.out.println(status);
55         }
56         else
57         {
58             status = "Status: Device supports SPP.";
59             System.out.println(status);
60             StreamConnection objectConnection = BluetoothConnection
61                 .getConnectionBluetooth(connectionUrl);
62             OutputStream outStream = objectConnection.openOutputStream();
63             InputStream inStream = objectConnection.openInputStream();
64             scanner = new ELM327(inStream, outStream);
65         }
66         return scanner;
67     }
```

Fonte: produzido pelo autor