

UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

OBD-JRP: MONITORAMENTO VEICULAR COM JAVA E
RASPBERRY PI

RICARDO ARTUR STAROSKI

BLUMENAU
2016

RICARDO ARTUR STAROSKI

OBD-JRP: MONITORAMENTO VEICULAR COM JAVA E RASPBERRY PI

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

Prof. Miguel Alexandre Wisintainer - Orientador

**BLUMENAU
2016**

OBD-JRP: MONITORAMENTO VEICULAR COM JAVA E RASPBERRY PI

Por

RICARDO ARTUR STAROSKI

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

Presidente:

Prof. Miguel Alexandre Wisintainer, Mestre – Orientador, FURB

Membro:

Profª. Luciana Pereira de Araújo, Mestre – FURB

Membro:

Prof. Mauro Marcelo Mattos, Doutor – FURB

Blumenau, 08 de dezembro de 2016

Dedico este trabalho aos meus pais, pelo amor,
apoio e compreensão por toda a vida.

AGRADECIMENTOS

Aos meus pais, pelo amor, apoio e compreensão, perante qualquer dificuldade, durante toda a vida.

À empresa Senior Sistemas, por flexibilizar meus horários de trabalho e disponibilizar sua infraestrutura para a realização deste trabalho.

Ao amigo e colega de equipe, Dennis Hiebert, por compreender e aceitar minhas ausências na empresa, para a realização deste trabalho.

Ao amigo e colega de equipe, Stephan Dieter Bieging, por ceder seu veículo para os testes de campo na realização deste trabalho.

Ao amigo Norberto Jensen, por ceder seu modem 3G para os testes de campo na realização deste trabalho.

Ao amigo e professor Fabricio Vegini, pelo apoio, incentivo e conhecimento transmitido durante toda a vida.

Ao professor Miguel Alexandre Wisintainer, pela orientação, disponibilidade e entusiasmo em me ajudar e pelo auxílio extraclasse durante a realização deste trabalho.

Ao professor Maurício Capobianco Lopes, por me despertar o fascínio pelo desenvolvimento de jogos e pelo auxílio extraclasse durante a realização deste trabalho.

Ao professor Roberto Heinzle, pelo entusiasmo com que sempre ministrou as aulas de estruturas de dados e pelo auxílio extraclasse durante a realização deste trabalho.

Ao professor Aurélio Faustino Hoppe, pela paciência e apoio prestado durante meu reingresso ao curso.

Ao professor Mauro Marcelo Mattos, por ter me motivado com o processador virtual que desenvolvi para me ajudar no aprendizado de arquitetura de computadores.

À professora Joyce Martins, pela dedicação e entusiasmo em ajudar os alunos nas suas disciplinas e por me despertar o fascínio por compiladores.

Ao professor Alexander Roberto Valdameri, por tornar bastante interessante as aulas de bancos de dados, assunto com o qual não tenho afinidade.

Agradeço ainda a todas as pessoas que direta ou indiretamente contribuíram com meu crescimento acadêmico, profissional e pessoal.

Forja o teu espírito como o de uma espada, do mais forte aço e com o melhor fio, pois dele dependerá a sua vida.

Masaaki Hatsumi

RESUMO

O presente trabalho tem por finalidade avaliar a viabilidade do desenvolvimento, em Java, de um protótipo de software embarcado em uma placa Raspberry Pi para monitorar os sensores de um veículo, através de comunicação com uma interface ELM327 Bluetooth conectada à porta On Board Diagnostic (OBD2) e disponibilizar estas informações em uma página web. O protótipo é dividido em duas partes distintas, o firmware instalado em um veículo e o servidor que disponibiliza páginas web para o monitoramento do veículo. O firmware e servidor são especificados utilizando o paradigma de orientação à objetos, utilizando diagramas da Unified Modeling Language (UML). A execução do protótipo ocorre de forma autônoma ao ser conectado em um adaptador USB automotivo. O protótipo é desenvolvido utilizando padrões de projeto de software. O resultado de sua construção é um dispositivo que permite o monitoramento via web e em tempo real do veículo.

Palavras-chave: Raspberry Pi. Internet das coisas. OBD2. IOT. Java. Bluetooth. Monitoramento veicular.

ABSTRACT

The present work has the purpose of evaluating the feasibility of developing, in Java, a software prototype embedded in a Raspberry Pi card to monitor the sensors of a vehicle by communicating with an ELM327 Bluetooth interface connected to the On Board Diagnostic (OBD2) port and making the information available on a web page. The prototype is divided into two distinct parts, the firmware installed in a vehicle and the server that provides web pages for monitoring the vehicle. The firmware and server are specified using the object orientation paradigm, along with Unified Modeling Language (UML). The prototype runs autonomously when plugged into an automotive USB adapter. The prototype is developed using software design standards. The result of its construction is a device that allows real-time web monitoring of the vehicle.

Key-words: Raspberry Pi. Internet of things. OBD2. IOT. Java. Bluetooth. Vehicular monitoring.

LISTA DE FIGURAS

Figura 1 - Conector SAE J1962 e respectiva pinagem.....	14
Figura 2 - Aspecto da interface ELM327 RS232	17
Figura 3 - Aspecto da interface ELM327 USB	17
Figura 4 - Aspecto da interface ELM327 Bluetooth	18
Figura 5 - Aspecto da interface ELM327 WiFi.....	18
Figura 6 - Blocos eletrônicos da interface ELM327.....	19
Figura 7 - Visão geral dos protocolos de comunicação OBD	20
Figura 8 - Características do Raspberry Pi 3 Model B	21
Figura 9 - Conectando PyOBD com o veículo	23
Figura 10 - Exibindo resultados de testes com PyOBD	23
Figura 11 - Verificando dados em tempo real com PyOBD	24
Figura 12 - Lendo e limpando códigos de falhas com PyOBD	24
Figura 13 - Velocidade do veículo no EnviroCar.....	25
Figura 14 - Velocidade média, trajeto e distância percorridos no EnviroCar	26
Figura 15 - Informações coletadas pelo EnviroCar durante o percurso	26
Figura 16 - Ciclo de vida do firmware	29
Figura 17 - Ciclo de vida do servidor.....	30
Figura 18 - Camadas e pacotes do firmware	31
Figura 19 - Relacionamento entre as principais classes do firmware.....	34
Figura 20 - Leitura de dados da interface ELM327 Bluetooth.....	36
Figura 21 - Envio dos dados pendentes	37
Figura 22 - Processamento de requisições no servidor.....	38
Figura 23 - Relacionamento entre as classes do servidor	39
Figura 24 - Versões do Sistema Operacional e Java no Raspberry Pi.....	40
Figura 25 - Arquitetura da API JABWT BlueCove	41
Figura 26 - Executando ObdJrpListDevices no Raspberry Pi.....	48
Figura 27 - Executando ObdJrpScanData no Raspberry Pi.....	49
Figura 28 - Interfaces ELM327 WiFi e Bluetooth	50
Figura 29 - Instalação no Volkswagen Gol 2010	72
Figura 30 - Instalação no Volkswagen SpaceFox 2009	72
Figura 31 - Aspecto do adaptador USB veicular	73

Figura 32 - Aspecto do mini modem USB	73
Figura 33 - Página inicial	74
Figura 34 - Página com leituras em tempo real	74
Figura 35 - Página com leituras gráficas em tempo real	75
Figura 36 - Página com histórico de leituras	75

LISTA DE QUADROS

Quadro 1 - Listando dispositivos e serviços Bluetooth	43
Quadro 2 - Disparando consulta de dispositivos com JABWT	44
Quadro 3 - DiscoveryListener para consulta de dispositivos	44
Quadro 4 - Classe Lock utilizada para sincronização de processos	45
Quadro 5 - Disparando consulta de serviços com JABWT	46
Quadro 6 - DiscoveryListener para consulta de serviços	47
Quadro 7 - Configuração de acesso Bluetooth	49
Quadro 8 - Método execute da classe ObdJrpScanData.....	51
Quadro 9 - Reestabelecendo conexão Bluetooth após erro	51
Quadro 10 - Métodos restartAfterError e stop da classe ObdJrpScanData	52
Quadro 11 - Atualizando interface de usuário em tempo real	52
Quadro 12 - Apresentando dados lidos na interface de usuário	53
Quadro 13 - Configuração dos Parsers	54
Quadro 14 - Construtor da classe Scanner.....	54
Quadro 15 - Inicializando objeto ELM327.....	55
Quadro 16 - Obtendo PIDs suportados pelo veículo	56
Quadro 17 - Processando máscara de bits	57
Quadro 18 - Classe EventMulticaster.....	58
Quadro 19 - Método execute da classe ScanLoop.....	59
Quadro 20 - Construtor da classe ScanLoop	59
Quadro 21 - Tratamento de eventos na classe ScanUploader	60
Quadro 22 - Método upload da classe ScanUploader.....	61
Quadro 23 - Método add da classe PackagePersister	62
Quadro 24 - Método execute da classe DataMonitor	62
Quadro 25 - Método upload da classe DataMonitor.....	63
Quadro 26 - Classe PostRequest.....	64
Quadro 27 - Classe ObdJrpServlet	65
Quadro 28 - Command SendData.....	66
Quadro 29 - Command UploadData.....	66
Quadro 30 - Command ReadData	67
Quadro 31 - Página vehicle-history.jsp	68

Quadro 32 - Página vehicle-detail.jsp.....	69
Quadro 33 - Command ViewChart.....	70
Quadro 34 - Página view-chart.jsp	71
Quadro 35 - Command ListVehicles	71
Quadro 36 - PIDs suportados do Gol e SpaceFox	77
Quadro 37 - Comparativo entre os trabalhos correlatos e o trabalho desenvolvido.....	78

LISTA DE ABREVIATURAS E SIGLAS

API – Application Program Interface

CARB – California Air Resources Board

CONAMA – Conselho Nacional do Meio Ambiente

CSI – Camera Serial Interface

DSI – Display Serial Interface

DTC – Diagnostic Trouble Code

ECU – Electronic Control Unit

GPIO – General Purpose Input/Output

GUI – Graphic User Interface

HDMI – High Definition Multimedia Interface

IOT – Internet of Things

JABWT – Java API for Bluetooth Wireless Technology

JNI – Java Native Interface

JSP – Java Server Pages

JSR – Java Specification Request

LIM – Lâmpada Indicadora de Mau Funcionamento

M2M – Machine to Machine

MIL – Malfunction Indicator Lamp

OBD – On Board Diagnostic

PC – Personal Computer

PID – Parameter Identification Number

RAM – Random Access Memory

SAE – Society of Automotive Engineers

SD – SanDisk

SPP – Serial Port Profile

UML – Unified Modeling Language

USB – Universal Serial Bus

UUID – Universally Unique Identifier

VIN – Vehicle Identification Number

SUMÁRIO

1 INTRODUÇÃO.....	10
1.1 OBJETIVOS DO TRABALHO	11
1.2 ESTRUTURA.....	11
2 FUNDAMENTAÇÃO TEÓRICA	12
2.1 HISTÓRIA DO OBD	12
2.2 PROTOCOLOS OBD2	14
2.3 MODOS DE DIAGNÓSTICO	15
2.4 INTERFACE ELM327.....	16
2.5 RASPBERRY PI	20
2.6 TRABALHOS CORRELATOS	22
2.6.1 PYOBD	22
2.6.2 ENVIROCAR	25
3 DESENVOLVIMENTO DO PROTÓTIPO	27
3.1 REQUISITOS.....	27
3.2 ESPECIFICAÇÃO	28
3.2.1 ESPECIFICAÇÃO DO FIRMWARE.....	30
3.2.2 ESPECIFICAÇÃO DO SERVIDOR	37
3.3 IMPLEMENTAÇÃO	40
3.3.1 Técnicas e ferramentas utilizadas.....	40
3.3.2 Operacionalidade da implementação	72
3.4 RESULTADOS E DISCUSSÕES.....	76
4 CONCLUSÕES.....	79
4.1 EXTENSÕES	79
REFERÊNCIAS	80

1 INTRODUÇÃO

A Internet das Coisas, ou Internet of Things (IOT), se refere a uma revolução tecnológica que tem como objetivo conectar os itens usados do dia a dia à rede mundial de computadores (ZAMBARDA, 2014). Cada vez mais surgem eletrodomésticos, meios de transporte e até mesmo roupas conectadas à internet e a outros dispositivos, como computadores e smartphones. Segundo GSM Association (2014), soluções Machine to Machine (M2M), já utilizam redes sem fio para conectar dispositivos uns aos outros e à internet, com o mínimo de intervenção humana. A IOT é uma evolução do M2M e representa a coordenação de máquinas, dispositivos e aparelhos de vários fornecedores conectados à internet através de múltiplas redes (GSM ASSOCIATION, 2014, tradução nossa).

Grande parte dos dispositivos domésticos incluem conectividade WiFi ou Bluetooth permitindo a comunicação com outros dispositivos e aparelhos (NG, 2015). Segundo Ng (2015), a capacidade de realizar análises em tempo real mudou para sempre a IOT, permitindo a implementação de sistemas preditivos e analíticos de forma eficiente. A principal aplicação dessas análises é auxiliar a identificar a causa raiz de falhas dos aparelhos, de forma a facilitar o processo de reparação (NG, 2015).

A especificação de um sistema capaz de recolher informações e estabelecer os diagnósticos de bordo é vantajosa para o dono do veículo, bem como para um técnico de reparação (ZURAWSKI, 2009, p. 33, tradução nossa). O termo utilizado para esta função chama-se diagnose de bordo ou On Board Diagnostic (OBD). O conceito OBD refere-se ao auto diagnóstico do estado dos componentes do veículo. Segundo Zurawski (2009), o OBD só se tornou possível devido à introdução de sistemas computadorizados nos veículos. O papel das funções de diagnóstico predecessoras ao OBD era limitado a piscar uma luz assim que um problema específico fosse detectado. Zurawski (2009) explica que os sistemas OBD recentes são baseados na padronização da comunicação, dos dados monitorados e dos códigos de uma lista de falhas específicas.

CONAMA (2004) considera que o OBD, constitui tecnologia de ação comprovada na identificação de mau funcionamento de um veículo. Segundo CONAMA (2004), através da análise dos dados, é possível prevenir a ocorrência de avarias dos componentes do veículo.

Diante do exposto, este trabalho consiste no desenvolvimento de um protótipo de software embarcado em uma placa Raspberry Pi, para coletar informações da porta OBD de um veículo e disponibilizar estas informações em uma página web.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é o desenvolvimento de um software embarcado, para coletar os dados da porta OBD2 de um automóvel e enviá-los para um servidor web.

Os objetivos específicos do trabalho são:

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

1.2 ESTRUTURA

O trabalho está organizado em quatro capítulos. O capítulo 2 descreve a fundamentação teórica utilizada para embasar este trabalho. É apresentada a história do OBD, os protocolos OBD2, a interface ELM327 e o computador Raspberry Pi. O capítulo é finalizado com os trabalhos correlatos.

O capítulo 3 traz a especificação do firmware e servidor e detalhes da implementação de ambos. São apresentados detalhes da operacionalidade do protótipo. Ao final do são analisados os resultados do desenvolvimento e execução.

O capítulo 4 traz a conclusão do trabalho juntamente com sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem como objetivo explorar os principais assuntos necessários para a realização deste trabalho. Os assuntos foram subdivididos em seis partes, onde a seção 2.1 apresenta a história do OBD. A seção 2.2 expõe os protocolos OBD2. A seção 2.3 apresenta os modos de diagnóstico. A seção 2.4 apresenta a interface ELM327. A seção 2.5 apresenta a plataforma Raspberry Pi e, por fim, na seção 2.6 são descritos dois trabalhos correlatos.

2.1 HISTÓRIA DO OBD

On Board Diagnostic (OBD) significa Diagnóstico de Bordo. Este diagnóstico é realizado pelas próprias unidades eletrônicas do veículo. Segundo Manavella (2009), em 1988 o California Air Resources Board (CARB), estabeleceu uma norma não padronizada denominada OBD1 para que todos os veículos vendidos no estado da Califórnia, nos EUA, incorporassem em sua unidade de comando um sistema de diagnóstico capaz de detectar defeitos nos elementos e sistemas de controle de emissões. Manavella (2009) complementa que o OBD1 especificava um indicador luminoso chamado Malfunction Indicator Lamp (MIL), que acendia na presença de falhas. No Brasil o indicador MIL é chamado de Lâmpada Indicadora de Mau Funcionamento (LIM) (CONAMA, 2004).

No Brasil, o Conselho Nacional do Meio Ambiente (CONAMA), determinou a introdução dos sistemas de diagnose de bordo, em duas etapas complementares e consecutivas denominadas OBDBr-1 e OBDBr-2. De acordo com CONAMA (2004), o sistema OBDBr-1 foi implantado em sua totalidade em 1º de janeiro de 2009 e definiu as características mínimas para a detecção de falhas nos seguintes componentes, quando aplicável:

- a) sensor de pressão absoluta ou fluxo de ar;
- b) sensor de posição da borboleta;
- c) sensor de temperatura de arrefecimento;
- d) sensor de temperatura de ar;
- e) sensor de oxigênio;
- f) sensor de velocidade do veículo;
- g) sensor de posição do eixo comando de válvulas;
- h) sensor de posição do virabrequim;
- i) sistemas de recirculação dos gases de escape;
- j) sensor para detecção de detonação;
- k) válvulas injetoras;
- l) sistema de ignição;

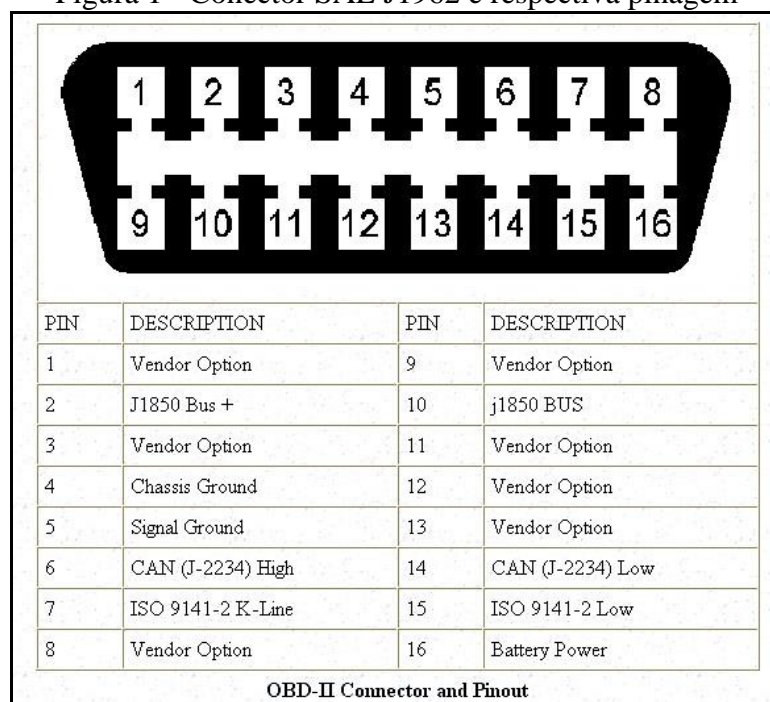
- m) módulo controle eletrônico do motor;
- n) lâmpada indicadora de mau funcionamento;
- o) outros componentes que o fabricante julgue relevantes para a correta avaliação do funcionamento do veículo e controle de emissões de poluentes.

CONAMA (2004) considera que o sistema OBDBr-2 complementa as funções e características do sistema OBDBr-1. Segundo CONAMA (2004), o sistema OBDBr-2 deve detectar e registrar a existência de falhas, deterioração dos sensores de oxigênio e eficiência de conversão do catalisador. CONAMA (2004) complementa que o sistema OBDBr-2 deve apresentar características mínimas para a detecção de falhas nos seguintes componentes, quando aplicável:

- a) sensores de oxigênio (pré e pós-catalisador);
- b) eletroválvula do cânister;
- c) outros componentes que o fabricante julgue relevantes para a correta avaliação do funcionamento do veículo e controle de emissões de poluentes.

Não demorou muito para o CARB concluir que o padrão OBD1 não era eficiente para determinar o elemento que provocara o defeito. Portanto o CARB desenvolveu um novo conjunto de especificações, surgindo assim a norma OBD2 (MANAVELLA, 2009, p. 121). A Society of Automotive Engineers (SAE), estabeleceu a norma SAE J1962, que determinou o conector J1962 fêmea de 16 pinos, como a interface de hardware padrão para o OBD2. Na Figura 1 observa-se o aspecto e pinagem do conector J1962 (SAE INTERNATIONAL, 2006).

Figura 1 - Conector SAE J1962 e respectiva pinagem



Fonte: RioRand (2015).

Além do conector físico, a SAE também estabeleceu a norma SAE J1979, que define o método de requisição de dados de diagnóstico e uma lista dos parâmetros padrões disponíveis na Electronic Control Unit (ECU) (SAE INTERNATIONAL, 2006). Cada parâmetro é denominado Parameter Identification Number (PID) e cada código de erro é denominado Diagnostic Trouble Code (DTC). Conforme SAE International (2006), não é exigido que os fabricantes implementem todos os PIDs, é permitido a inclusão de PIDs proprietários, não listados na norma SAE J1979 e é permitido o acesso em tempo real aos PIDs e DTCs do veículo.

2.2 PROTOCOLOS OBD2

Enquanto a porta OBD2 é normalizada em todo o mundo, vários protocolos de comunicação continuam possíveis, dependendo dos fabricantes de veículos (TOTAL CAR, 2014, tradução nossa). Atualmente estes protocolos podem ser classificados em três famílias: Redes CAN, Linhas K/L e SAE J1850.

Segundo Total Car (2014), redes CAN utilizam os pinos 6 e 14 do conector J1962 e compreendem os seguintes protocolos:

- a) ISO 157565: Utilizado por todos os veículos. Velocidade de comunicação de 125 a 500 Kbps;
- b) SAE J1939: Utilizado principalmente por veículos pesados como caminhões e máquina agrícolas e possui velocidade de comunicação de 125 a 500 Kbps.

Segundo Total Car (2014), Linhas K/L utilizam os pinos 7 e 15 do conector J1962 e compreendem os seguintes protocolos:

- a) ISO 9141-2: Utilizado principalmente por fabricantes europeus e possui velocidade de comunicação de 10,4 Kbps;
- b) ISO 14230 (KWP2000): Utilizado principalmente por fabricantes europeus. Dentro deste protocolo, existem dois sub protocolos que diferem no tempo de inicialização:
 - Slow init, “inicialização lenta” com velocidade de comunicação de 1,4 a 10,4 Kbps;
 - Fast init, “inicialização rápida” com velocidade fixa de 10,4 Kbps.

Segundo Total Car (2014), SAE J1850 compreende os seguintes protocolos:

- a) PWM: utilizado principalmente pela Ford Motors. Velocidade de comunicação de 41,6 Kbps. Utiliza os pinos 2 e 10 do conector J1962;
- b) VPW: utilizado principalmente pela General Motors. Velocidade de comunicação de 10,4 a 41,6 Kbps. Utiliza somente o pino 2 do conector J1962.

2.3 MODOS DE DIAGNÓSTICO

Independente do protocolo utilizado, o padrão OBD2 define 10 modos de diagnóstico, são eles:

- a) modo 1: Retorna valores comuns de alguns sensores como por exemplo, rotações do motor, velocidade do veículo, temperatura do motor, sensores de oxigênio e mistura ar/combustível. Cada sensor é identificado por um PID;
- b) modo 2: Obtém o “instantâneo” de uma falha. Quando a ECU detecta uma falha, ela grava os dados do sensor daquele momento específico;
- c) modo 3: Apresenta os DTCs armazenados. Segundo Outils OBD Facile (2015), estes códigos são padrão para todas as marcas de veículos e são divididos em quatro categorias:
 - P0xxx: Para falhas associadas ao motor e transmissão;
 - C0xxx: Para falhas associadas ao chassi;
 - B0xxx: para falhas associadas à carroceria;
 - U0xxx: para falhas associadas à comunicação de rede.
- d) modo 4: Utilizado para apagar os DTCs gravados e desligar o MIL;

- e) modo 5: Retorna o autodiagnostico do sensor lambda¹. Segundo Outils OBD Facile (2015), este modo não é mais utilizado pois o modo 6 substitui suas funções;
- f) modo 6: Retorna os resultados do autodiagnostico realizado nos diversos sensores do veículo;
- g) modo 7: Este modo retorna DTCs não confirmados. Segundo Outils OBD Facile (2015), isto é bastante útil após um reparo no veículo, para confirmar que um DTC não está mais presente. Seus códigos são idênticos aos do modo 3;
- h) modo 8: Segundo The Best OBD2 Scanners (2016), diferente dos outros modos que servem somente para ler informações, este modo é bidirecional, permitindo também gravar informações;
- i) modo 9: Este modo obtém informações do veículo como por exemplo seu número de identificação;
- j) modo 10: Este modo obtém os DTCs permanentes que, diferente dos modos 3 e 7, não podem ser apagados utilizando o modo 4. Outils OBD Facile (2015) explica que estes DTCs são apagados automaticamente pela própria ECU após rodar vários quilômetros sem que se repitam.

Não necessariamente todos os modos são suportados pelas ECUs. Quanto mais recente for o veículo, maior é a chance de haver suporte a mais modos (OUTILS OBD FACILE, 2015).

2.4 INTERFACE ELM327

Segundo Total Car (2014), existem vários tipos de interface OBD2 e as mais comuns utilizam o circuito ELM327. De acordo com ELM Electronics (2016), o circuito ELM327 suporta todos os protocolos OBD2. Total Car (2014) explica que existem 4 tipos de interface ELM327:

- a) ELM327 RS232: Conexão serial que está gradativamente desaparecendo nos computadores modernos. A Figura 2 apresenta o aspecto desta interface;
- b) ELM327 USB: Conexão Universal Serial Bus (USB), presente na maioria dos computadores atuais. A Figura 3 apresenta o aspecto desta interface;
- c) ELM327 Bluetooth: Conexão sem fio, que pode ser utilizada com computadores ou smartphones. A Figura 4 apresenta o aspecto desta interface;
- d) ELM327 WiFi: Conexão sem fio que pode ser utilizada com computadores ou

¹ Horta (2000) explica que o sensor lambda é responsável pelo ajuste fino da mistura ar-combustível.

smartphones. Aspecto idêntico ao da interface Bluetooth, como ilustra a Figura 5.

Figura 2 - Aspecto da interface ELM327 RS232



Fonte: Total Car (2014).

Figura 3 - Aspecto da interface ELM327 USB



Fonte: Total Car (2014).

Figura 4 - Aspecto da interface ELM327 Bluetooth



Fonte: Total Car (2014).

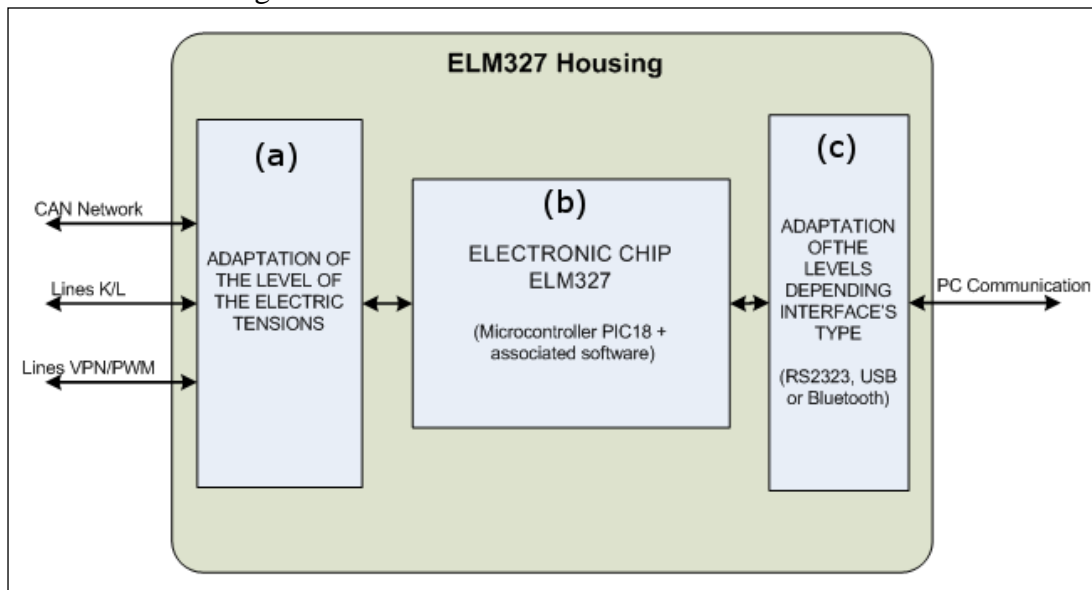
Figura 5 - Aspecto da interface ELM327 WiFi



Fonte: Total Car (2014).

Apesar das aparências, estas 4 interfaces são eletronicamente idênticas. Somente o seu aspecto externo e o tipo de conexão são diferentes. No seu interior reside um circuito ELM327 (TOTAL CAR, 2014, tradução nossa). Na Figura 6 são apresentados os blocos que compõe a interface ELM327:

Figura 6 - Blocos eletrônicos da interface ELM327



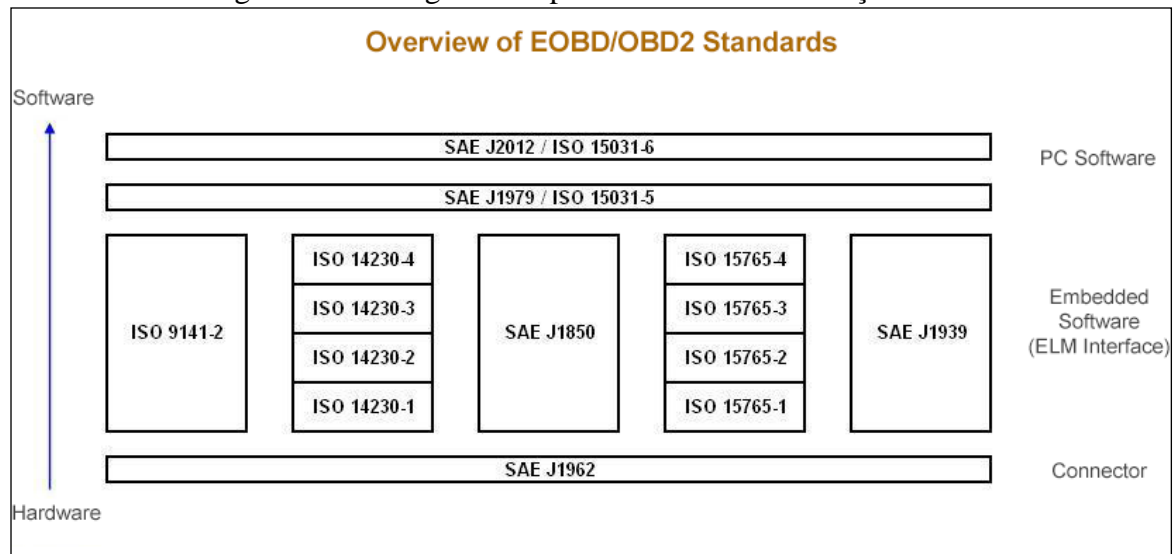
Fonte: Total Car (2014).

Na Figura 6 são apresentados os blocos que compõe a unidade eletrônica da interface ELM327, sendo eles:

- a) adaptadores de tensão elétrica: as redes on-board nos carros possuem níveis de tensão que requerem drivers específicos. Como o ELM327 suporta diversos protocolos, diversos drivers são necessários;
- b) chip ELM327: é o circuito integrado, cujo nome é aplicado ao dispositivo como um todo. Ele seleciona o protocolo e o converte para um protocolo reconhecido por modems de computador. Ele atua como uma ponte entre os protocolos;
- c) adaptadores de tensão para o computador: o chip por si só não é hábil para se comunicar com o computador, ele precisa adaptar os níveis de tensão antes de enviar o fluxo de dados.

Na Figura 7 observa-se a representação em colunas dos diversos protocolos suportados pela interface ELM327.

Figura 7 - Visão geral dos protocolos de comunicação OBD



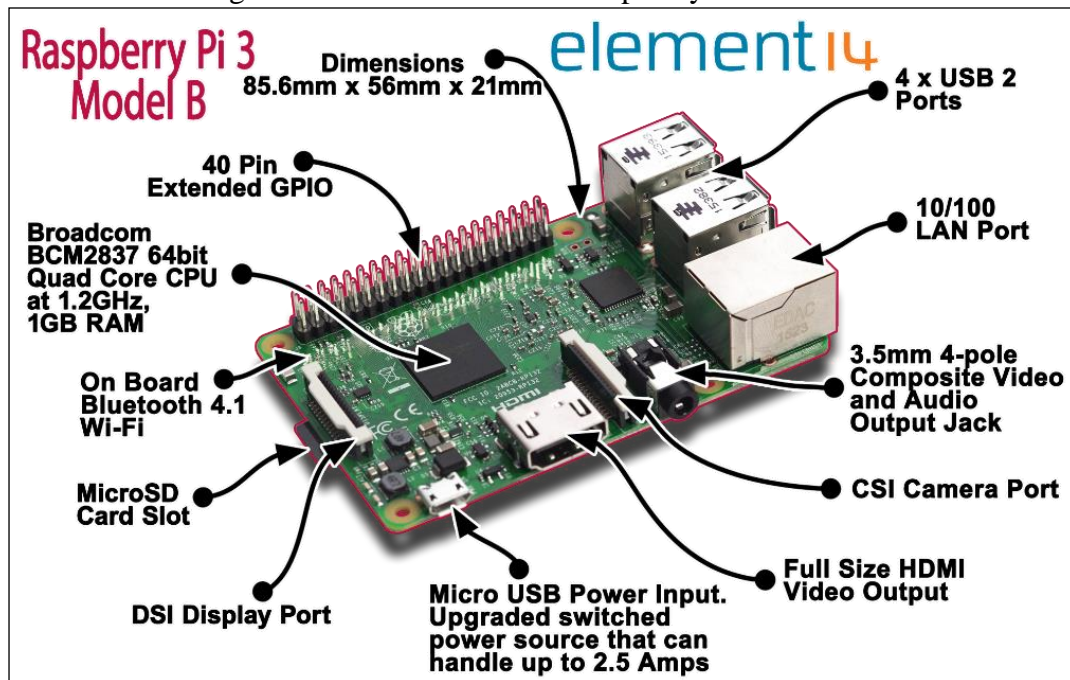
Fonte: Total Car (2014).

Segundo Total Car (2014), o papel do ELM327 é decodificar os diversos protocolos OBD2 (ISO 9141-2, ISO 14230, SAE J1850, ISO 15765 e SAE J1979). Desta forma o ELM327 abstrai os detalhes da comunicação com o hardware e simplifica a implementação do software.

2.5 RASPBERRY PI

O Raspberry Pi é um Personal Computer (PC), miniaturizado baseado no processador ARM. Ele pode realizar a maioria das tarefas que um desktop PC realiza, como por exemplo executar planilhas de cálculo, editores de texto e jogos (NEW IT LIMITED, 2016). Segundo Raspberry Pi Foundation (2016), ele foi desenvolvido para permitir que pessoas de todas as idades possam explorar a computação, aprender a programar e entender o funcionamento dos computadores. Na Figura 8 observa-se o aspecto da placa Raspberry Pi 3 Model B.

Figura 8 - Características do Raspberry Pi 3 Model B



Fonte: Thomsen (2016).

New IT Limited (2016) apresenta a seguinte especificação técnica da placa Raspberry Pi 3 Model B:

- a) computador de placa única com chipset Broadcom BCM2837;
- b) processador quad core ARM Cortex-A53 de 1,2GHz;
- c) 1GB de Random Access Memory (RAM);
- d) 40 pinos de General Purpose Input/Output (GPIO);
- e) conexão Bluetooth 4.1 integrada;
- f) conexão WiFi 802.11n integrada;
- g) 1 porta Ethernet 10/100;
- h) 4 portas USB;
- i) 1 conector de 4 polos, combinado para saída de áudio estéreo e vídeo composto;
- j) 1 saída High Definition Multimedia Interface (HDMI);
- k) 1 porta Camera Serial Interface (CSI);
- l) 1 porta Display Serial Interface (DSI);
- m) 1 porta micro SanDisk (SD), para carga do sistema operacional e armazenamento de dados;
- n) 1 porta micro USB para fonte de alimentação.

2.6 TRABALHOS CORRELATOS

A seguir serão apresentados dois trabalhos correlatos ao trabalho desenvolvido. O item 2.6.1 apresenta o PyOBD, uma ferramenta de diagnóstico automotivo compatível com OBD2 desenvolvida em linguagem de programação Python (PYOBD, 2015). O item 2.6.2 apresenta o EnviroCar, um aplicativo que permite compartilhar informações obtidas através da porta OBD2 (ENVIROCAR, 2015).

2.6.1 PYOBD

Trata-se de uma ferramenta open source de diagnóstico automotivo, segundo PyOBD (2015), a ferramenta foi projetada para se conectar à porta OBD2 através de uma interface ELM327 USB. O PyOBD é voltado para desenvolvedores Python, é composto de um único módulo, chamado `obd_io`, que permite um controle de alto nível sobre os dados dos sensores e gerenciamento dos códigos de erro (PYOBD, 2015). De acordo com PyOBD (2015), o módulo `obd_io` foi testado para funcionar em notebooks ou desktop PCs com os sistemas operacionais Microsoft Windows, Linux e Mac OSX. Seus pré-requisitos são:

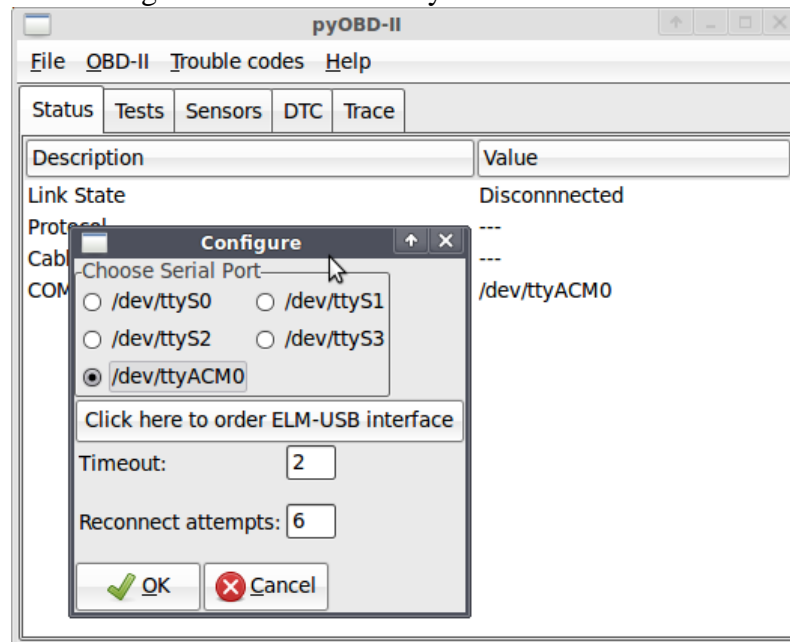
- a) uma interface ELM327 USB;
- b) python 2.x ou superior;
- c) pacote `py_serial`;
- d) um veículo que implemente o padrão OBD2.

Com o PyOBD é possível:

- a) conectar-se ao veículo;
- b) exibir resultados de testes;
- c) verificar dados dos sensores em tempo real;
- d) ler e limpar códigos de falhas DTC.

Na Figura 9 é apresentada a tela de conexão com o veículo. Observa-se que é possível selecionar a porta serial, o timeout e o número de tentativas para conectar-se.

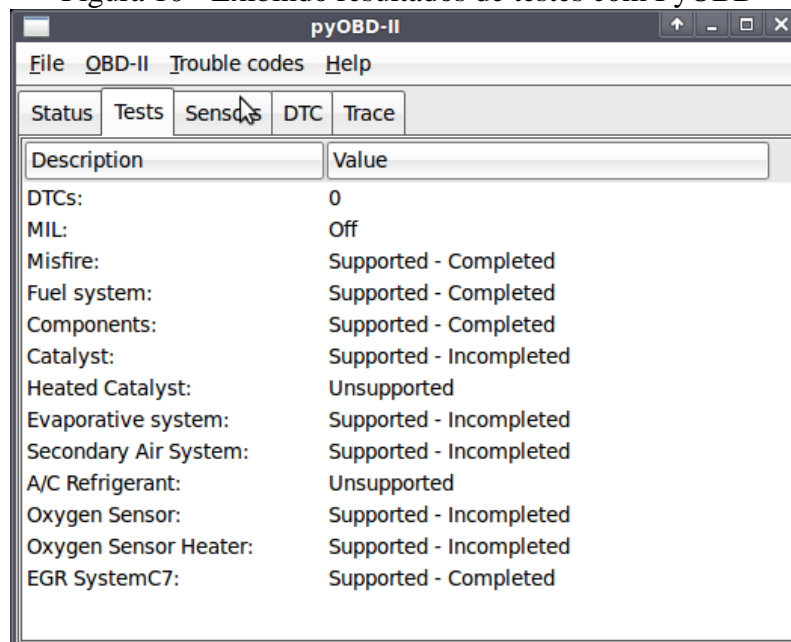
Figura 9 - Conectando PyOBD com o veículo



Fonte: PyOBD (2015).

Na Figura 10 é apresentada a tela de testes. Observa-se os diversos testes suportados e não suportados pelo veículo.

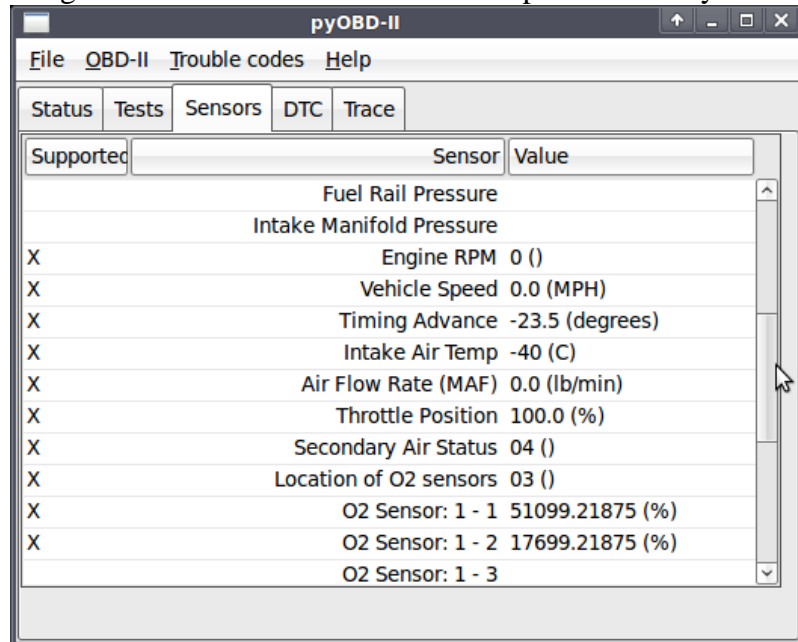
Figura 10 - Exibindo resultados de testes com PyOBD



Fonte: PyOBD (2015).

Na Figura 11 é apresentada a tela de leitura em tempo real. Observa-se os diversos parâmetros suportados pelo veículo e o respectivo valor lido.

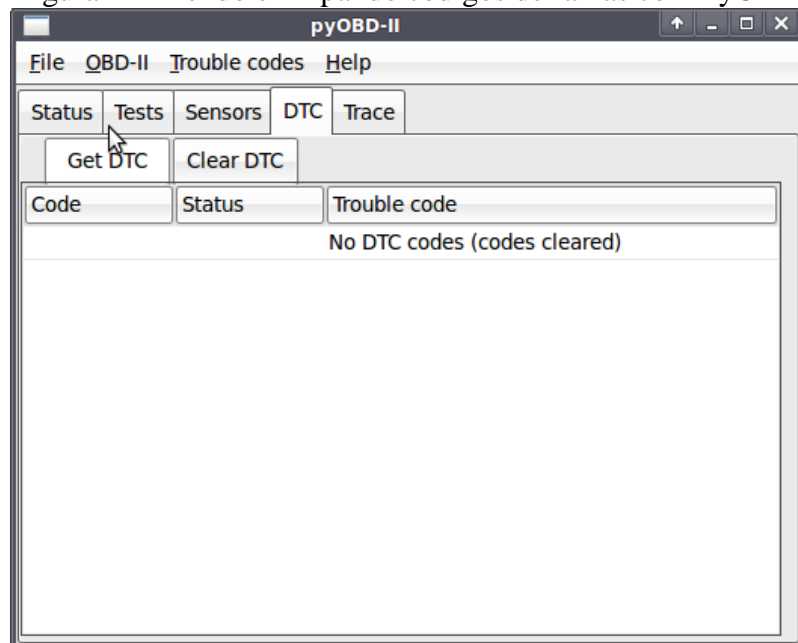
Figura 11 - Verificando dados em tempo real com PyOBD



Fonte: PyOBD (2015).

Na Figura 12 é apresentada a tela de códigos de falha. Observa-se que é possível ler os códigos de falha através do botão `Get DTC` e limpar os códigos através do botão `Clear DTC`.

Figura 12 - Lendo e limpando códigos de falhas com PyOBD



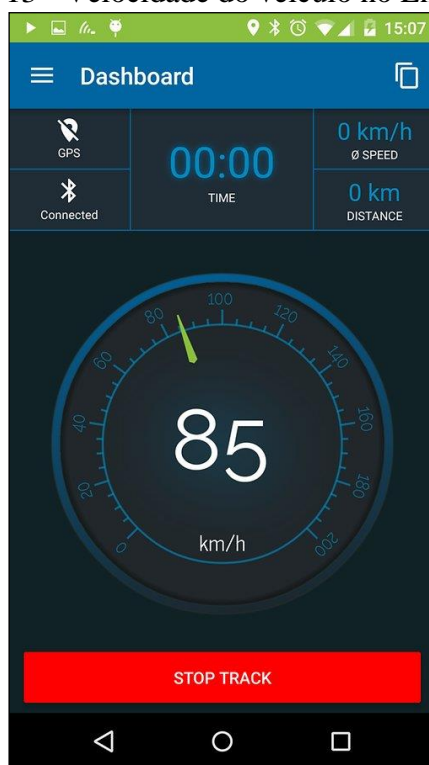
Fonte: PyOBD (2015).

2.6.2 ENVIROCAR

Trata-se de um aplicativo alemão open source, desenvolvido para smartphones Android. Seu propósito é que cidadãos, cientistas, engenheiros de tráfego e indústrias analisem dados OBD2 e compartilhem suas descobertas (ENVIROCAR, 2015, tradução nossa). O aplicativo conecta-se à porta OBD2 através de uma interface ELM327 Bluetooth. O usuário pode fazer upload das informações obtidas pelo aplicativo, diretamente para o servidor do EnviroCar. Segundo EnviroCar (2015), os dados ficam disponíveis anonimamente para que cientistas ou especialistas em tráfego acessem estes dados e os utilizem para solucionar questões ambientais e de mobilidade. EnviroCar permite que o usuário perceba o impacto ambiental causado pela forma de dirigir, investigando os dados dos sensores como consumo de combustível, emissão de gás carbônico e de ruídos (ANDROID PIT INTERNATIONAL, 2016).

A seguir são apresentadas algumas telas do aplicativo EnviroCar. Na Figura 13 observa-se a velocidade do veículo em quilômetros por hora.

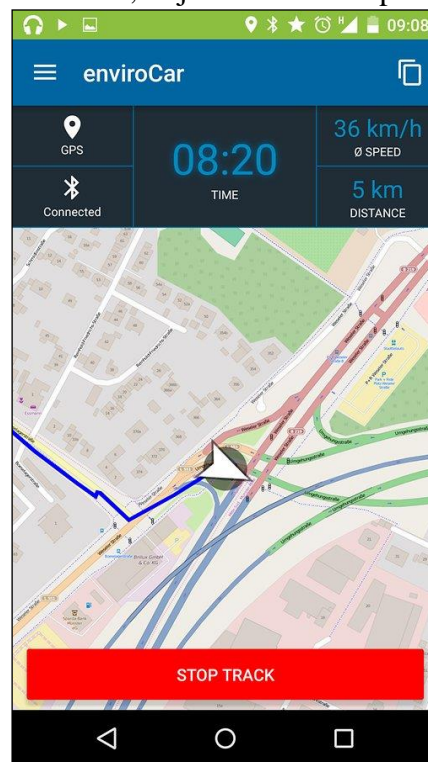
Figura 13 - Velocidade do veículo no EnviroCar



Fonte: Android Pit International (2016).

A Figura 14 apresenta um mapa com o desenho do trajeto percorrido, o tempo da viagem, a distância percorrida e a velocidade média.

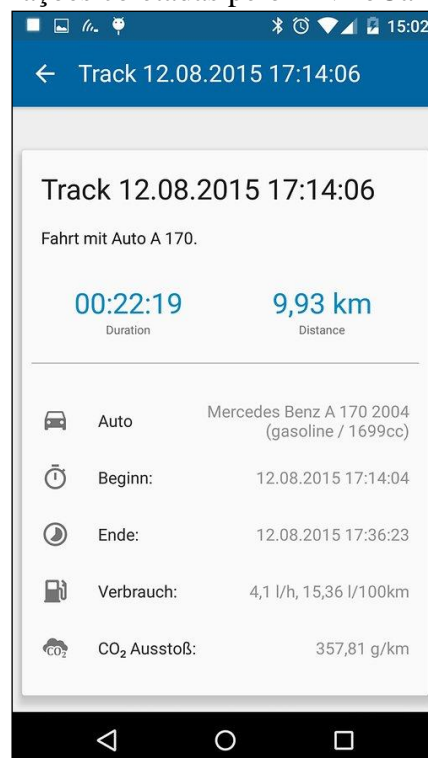
Figura 14 - Velocidade média, trajeto e distância percorridos no EnviroCar



Fonte: Android Pit International (2016).

Na Figura 15 são apresentadas diversas informações coletadas durante o percurso: marca e modelo do veículo, data e hora do início e término da viagem, consumo de combustível e a emissão de gás carbônico.

Figura 15 - Informações coletadas pelo EnviroCar durante o percurso



Fonte: Android Pit International (2016).

3 DESENVOLVIMENTO DO PROTÓTIPO

Neste capítulo são descritos os requisitos, a especificação do firmware e do servidor. Também é apresentada a implementação detalhando a operacionalidade do protótipo e os testes realizados em veículos reais. O capítulo finaliza com a descrição dos resultados obtidos.

3.1 REQUISITOS

Para simplificar a legibilidade, na descrição dos requisitos será utilizado o termo firmware para referenciar o software executando na placa Raspberry Pi instalada no veículo e o termo servidor para referenciar o software executando no servidor de aplicações TomCat. Os requisitos do protótipo a ser desenvolvido são:

- a) o firmware deve ser inicializado automaticamente ao ligar a placa Raspberry Pi (Requisito Funcional – RF);
- b) o firmware deve se conectar à porta OBD2 através de uma interface ELM327 Bluetooth (RF);
- c) o firmware deve coletar os dados da porta OBD2 e armazená-los localmente até serem enviados ao servidor (RF);
- d) o firmware deve tentar estabelecer uma conexão com o servidor a cada 5 minutos, caso não esteja conectado à internet (Requisito Não Funcional – RNF);
- e) o firmware deve enviar ao servidor o número do chassi do carro e os dados OBD2 armazenados localmente desde a última conexão bem-sucedida (RF);
- f) o firmware deve ser desenvolvido utilizando tecnologia Java SE (RNF);
- g) o firmware deve executar em sistema operacional Raspbian (RNF);
- h) o servidor deve responder requisições HTTP, através dos métodos GET e POST² (RF);
- i) o servidor deve persistir os dados coletados pelo firmware (RF);
- j) o servidor deve persistir os dados em arquivos XML, sem a necessidade de utilizar banco de dados (RNF);
- k) o servidor deve dispor uma página web para consultar os dados OBD2 a partir do número do chassi do carro (RF);
- l) o servidor deve ser desenvolvido utilizando tecnologia Java EE (RNF);
- m) o servidor deve executar no servidor de aplicações Apache TomCat (RNF);

² W3Schools (2016) explica que o método GET serve para requisitar dados de um determinado recurso e o método POST serve para enviar dados para serem processados por determinado recurso.

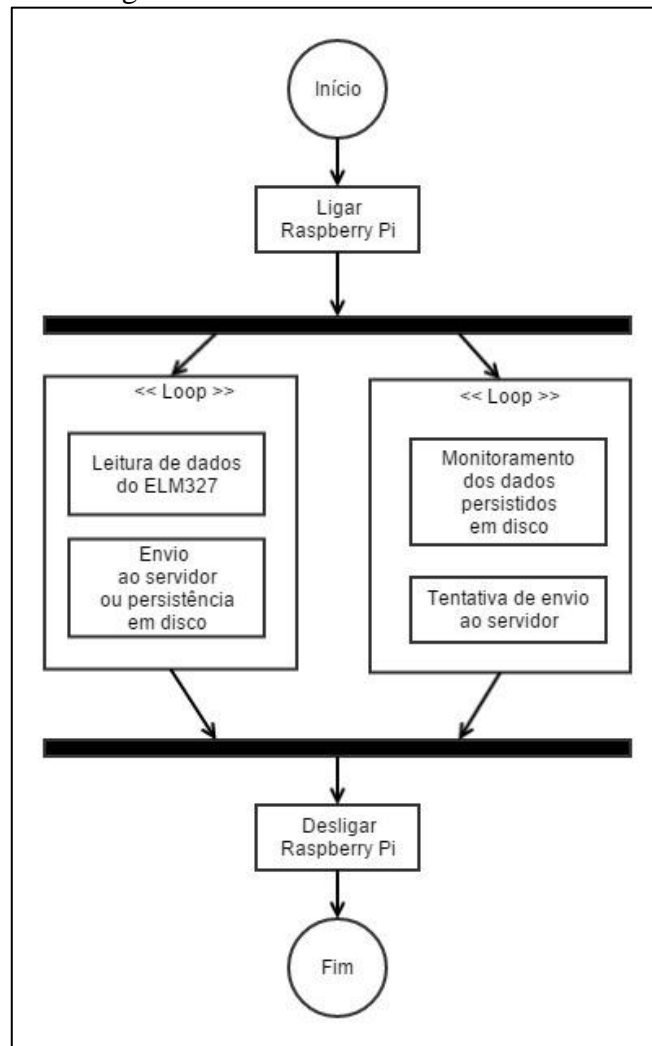
- n) a página web deve apresentar gráficos com os valores dos dados coletados (RF);
- o) a página web deve apresentar uma tabela com os valores dos dados coletados (RF);
- p) a página web deve ter interface responsiva de modo que possa ser visualizada em smartphones (RNF);
- q) a página web deve ser desenvolvida utilizando HTML, CSS e JavaScript (RNF).

3.2 ESPECIFICAÇÃO

A solução consiste no desenvolvimento de um firmware embarcado em uma placa Raspberry Pi que se comunica com uma interface ELM327 Bluetooth para obter dados OBD2 e com um servidor para o qual estes dados são enviados. Inicialmente será apresentada a especificação do firmware e posteriormente a especificação do servidor.

O ciclo de vida do firmware consiste em disparar dois processos paralelos após a inicialização do Raspberry Pi, sendo um processo responsável pela leitura dos dados da interface ELM327 Bluetooth e o outro processo responsável por monitorar o diretório onde os pacotes com os dados das leituras são persistidos. A Figura 16 apresenta o diagrama correspondente ao ciclo de vida do firmware.

Figura 16 - Ciclo de vida do firmware

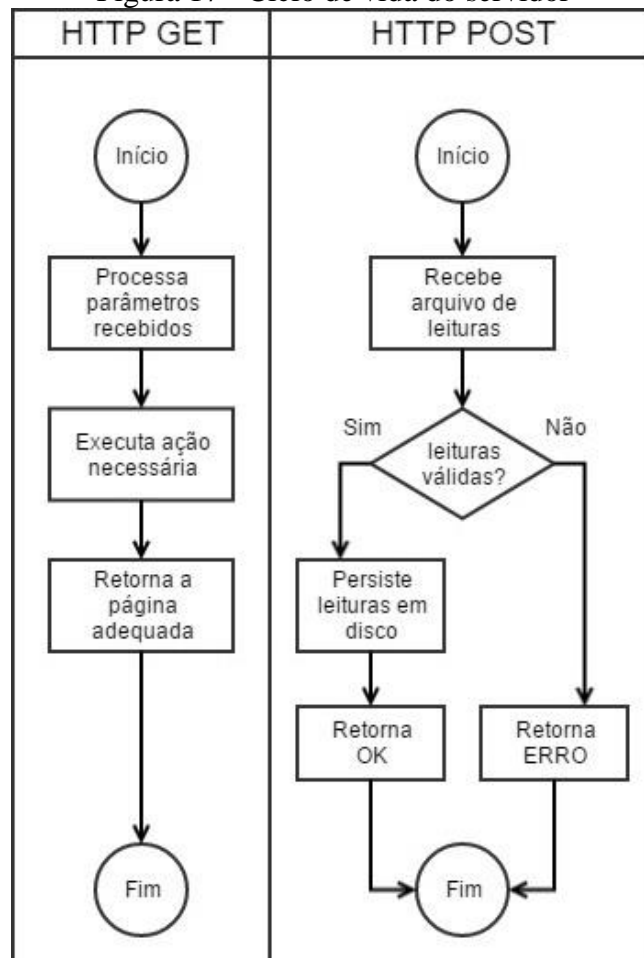


Fonte: Elaborado pelo autor.

Observa-se na Figura 16 dois processos distintos sendo disparados ao ligar o firmware. O primeiro processo executa continuamente a leitura dos dados da interface ELM327 e tenta enviar os mesmos ao servidor web. O segundo processo monitora os dados que não foram enviados com sucesso ao servidor e tenta enviá-los novamente. O ciclo dos dois processos somente é interrompido ao desligar o firmware.

O ciclo de vida do servidor consiste em processar requisições HTTP GET para retornar páginas solicitadas e processar requisições HTTP POST para receber os arquivos com leituras de dados enviadas pelo firmware. A Figura 17 apresenta o diagrama de atividades correspondente ao ciclo de vida do servidor.

Figura 17 - Ciclo de vida do servidor



Fonte: Elaborado pelo autor.

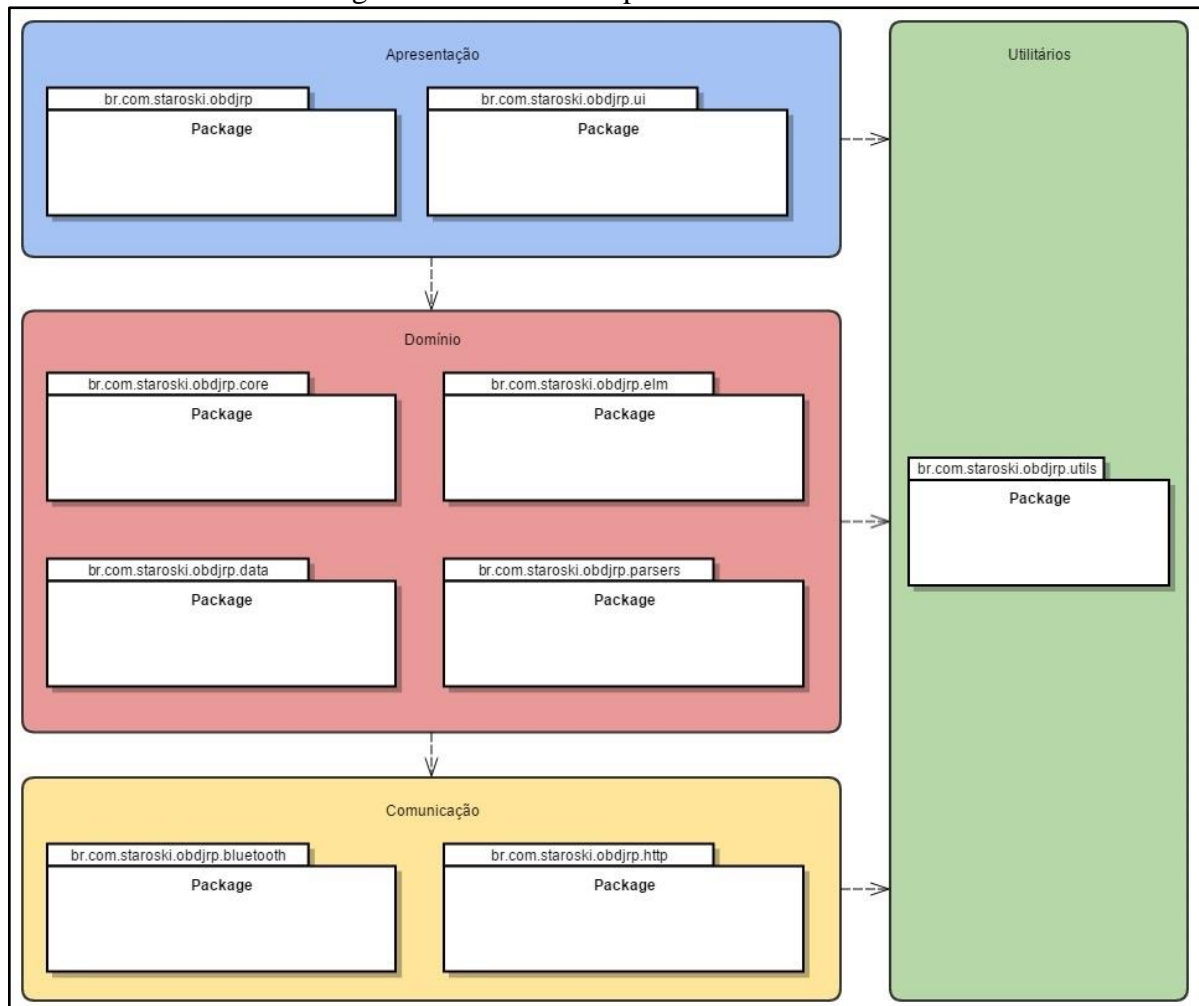
Observa-se na Figura 17 as diferentes ações tomadas pelo servidor ao receber requisições GET e POST. Ao receber uma requisição GET, os parâmetros da requisição são processados, a ação necessária é executada e em seguida a página adequada é retornada. Ao receber uma requisição POST, é processado o arquivo de leituras enviado na requisição, se o arquivo foi validado e persistido em disco, o servidor retorna a mensagem OK, caso contrário retorna a mensagem ERRO.

3.2.1 ESPECIFICAÇÃO DO FIRMWARE

O desenvolvimento do firmware foi dividido em 4 camadas distintas: apresentação, domínio, comunicação e utilitários. A camada de apresentação é responsável por fornecer o ponto de entrada para a execução dos processos no firmware e a interface de usuário, onde as leituras podem ser acompanhadas em tempo real. A camada domínio, como o nome sugere, contém as classes de domínio do protótipo. A camada de comunicação fornece classes que permitem a comunicação via Bluetooth, realização requisições HTTP GET e POST e na camada de utilitários residem classes de propósito geral, utilizadas pelas outras camadas. Na Figura 18

são apresentadas as camadas do firmware e a dependência entre eles, observa-se ainda quais os pacotes que compõe cada camada.

Figura 18 - Camadas e pacotes do firmware



Fonte: Elaborado pelo autor.

O pacote, `br.com.staroski.obdjrj` contém as classes que funcionam como ponto de entrada para a execução dos programas. Na linguagem Java isso corresponde às classes que declaram um método com a assinatura `public static void main(String[])`. Neste pacote estão definidas três classes:

- `ObdJrpListDevices`, programa que lista os dispositivos Bluetooth e seus serviços disponíveis;
- `ObdJrpScanData`, programa que lê os dados da ECU através de uma interface ELM327 Bluetooth, tentando enviá-los ao servidor ou persistindo-os em disco;
- `ObdJrpUploadData`, programa que monitora o diretório onde estão persistidas as leituras que o `ObdJrpScanData` não conseguiu enviar ao servidor e tenta reenviar estas leituras.

O pacote `br.com.staroski.obdjrp.ui`, contém a classe `ScannerWindow`, que representa uma Graphic User Interface (GUI), onde são apresentados em tempo real os dados lidos pelo programa `ObdJrpScanData`.

No pacote `br.com.staroski.obdjrp.core` estão as classes principais da API desenvolvida, são elas:

- a) `Config`, classe que implementa o padrão de projeto Singleton³ e representa a configuração dos programas, que é realizada através de um arquivo texto chamado `obd-jrp.properties`;
- b) `DataMonitor`, classe que monitora o diretório onde ficam os dados pendentes de envio e tenta enviá-las ao servidor, quem faz uso dessa classe é o programa `ObdJrpUploadData`;
- c) `IO`, interface para objetos compostos de um `InputStream` para leitura de dados e um `OutputStream` para escrita de dados, é utilizada como parâmetro de construção para objetos do tipo `Scanner`;
- d) `Scanner`, classe que realiza a leitura dos dados da ECU, comunicando-se com a interface ELM327, esta classe implementa o padrão de projeto Observer⁴, sendo possível registrar objetos para serem notificados quando as leituras são concluídas ou quando ocorrem erros;
- e) `ScannerListener`, interface para os objetos que desejam receber notificações do `Scanner`.

As classes e interfaces supracitadas são somente as classes públicas declaradas no pacote, entretanto nele ainda estão declaradas outras seis classes não públicas: `BluetoothIO`, `EventMulticaster`, `PackagePersister`, `ScanLoop`, `ScanUploader` e `SocketIO`. Estas classes somente são utilizadas pelas classes públicas do pacote, de forma a aumentar a granularidade da implementação segregando as classes em partes menores de responsabilidade específica.

O pacote `br.com.staroski.obdjrp.elm` contém duas classes públicas, `ELM327` e `ELM327Error`. A classe `ELM327`, como o nome sugere, representa uma interface ELM327, é através dela que são enviados os comandos ao hardware conectado ao veículo. A classe `ELM327Error` representa um erro que pode ser lançado pela classe `ELM327`. Neste pacote existe

³ Rocha (2005, p. 52) explica que o padrão Singleton garante que uma classe só tenha uma única instância, e provê um ponto de acesso global a ela.

⁴ Segundo Rocha (2005), a padrão Observer define uma dependência um-para-muitos entre objetos para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente.

ainda uma classe não pública chamada `Disconnecter`, que se registra à máquina virtual Java através do método `java.lang.Runtime.addShutdownHook` e quando a máquina virtual Java é encerrada, o `Disconnecter` itera sobre a lista de objetos ELM327 ativos e os desconecta.

No pacote `br.com.staroski.obdjrj.data` encontram-se as classes responsáveis pela representação, serialização e persistência dos dados lidos através da interface ELM327. As classes são as seguintes:

- a) `Data`, esta classe representa um dado lido da ECU e possui as propriedades `pid` e `value`, que contém o número do PID e os bytes lidos;
- b) `Scan`, esta classe representa um objeto de leitura, que contém uma lista com os objetos `Data` lidos durante um ciclo de leitura;
- c) `Package`, representa um pacote de dados pendentes de envio. Quando um `Scan` não é enviado com sucesso ao servidor, o mesmo é adicionado a um `Package`;
- d) `Parser`, é uma interface `Strategy`⁵ para objetos que implementam algoritmos que convertem objetos do tipo `Data` em um objetos do tipo `Parsed`;
- e) `Parsed`, representa um objeto `Data` que foi processado por um `Parser` de forma a obter informação humanamente legível e possui as propriedades `description` e `value`, que contém a descrição e o valor;
- f) `Parsing`, classe utilitária que utiliza implementações de `Parser` para transformar objetos `Data` em objetos `Parsed`.

O pacote `br.com.staroski.obdjrj.parsers` define algumas classes que implementam a interface `Parser`. Como supracitado, um objeto `Parser` é responsável por converter um objeto `Data` em um objeto `Parsed`. Considerando o cenário hipotético de um objeto `Data`, com as propriedades `pid=0C` e `value=0AF0`, este objeto corresponde a uma leitura das Rotações por Minuto (RPM)⁶. A classe `Parsing` será utilizada para obter um objeto `Parser` adequado ao PID do objeto `Data` e vai executar o algoritmo do cálculo de RPM⁷, este objeto `Parser` vai gerar um objeto `Parsed` com as propriedades `description="Engine RPM"` e `value=2800`, que é uma informação humanamente legível.

O pacote `br.com.staroski.obdjrj.bluetooth` define a classe `Bluetooth`, responsável por simplificar o acesso à API JABWT responsável pela descoberta de dispositivos

⁵ Conforme Rocha (2005), `Strategy` permite que algoritmos mudem independentemente entre clientes que os utilizam.

⁶ SAE International (2006, p. 118-190) define os PIDs e respectivos algoritmos para obter informação legível.

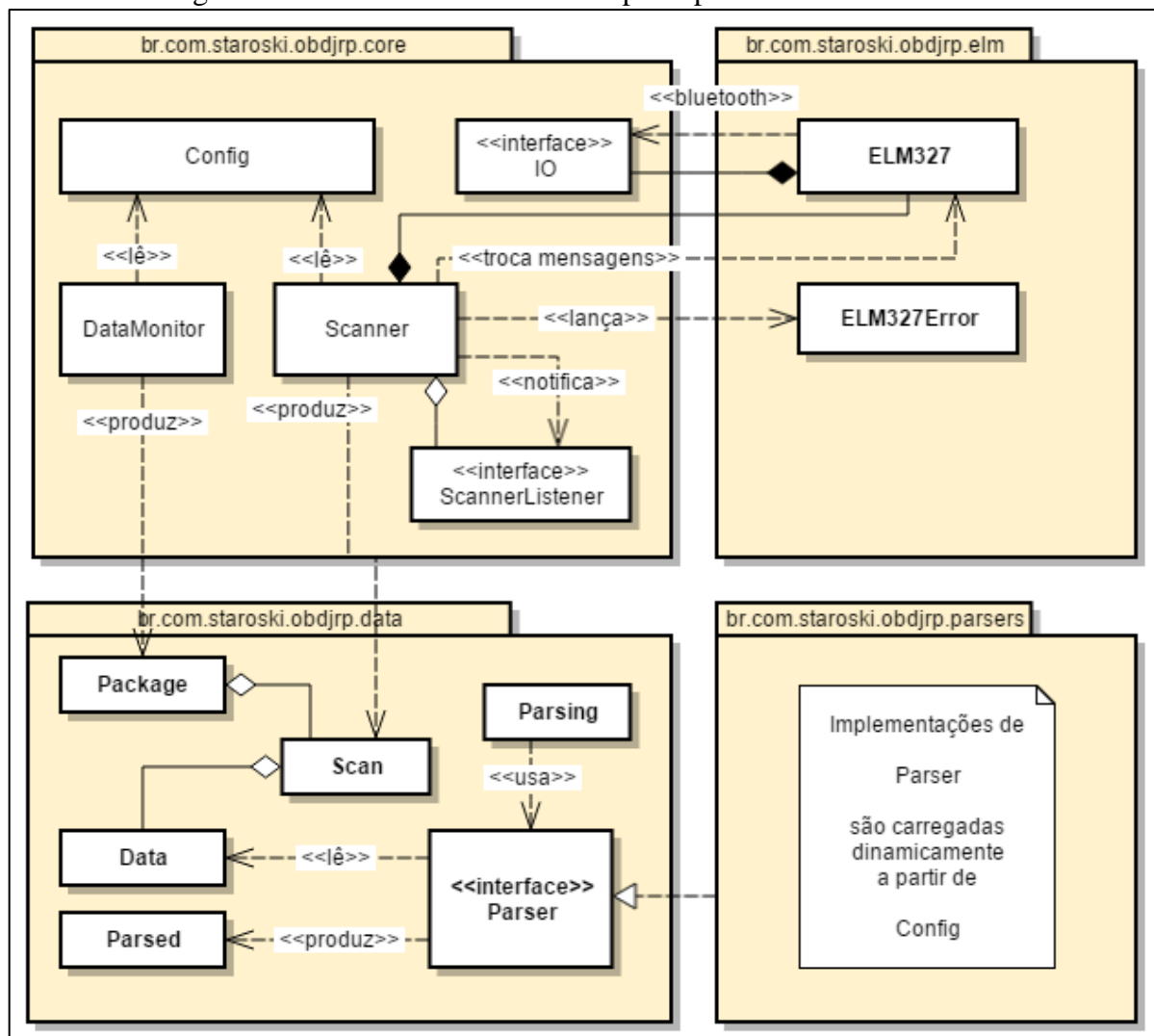
⁷ Conforme SAE International (2006, p. 129), obtém-se as RPM, dividindo valor decimal dos bytes por 4.

Bluetooth e conexão aos mesmos. A classe `Bluetooth` implementa o padrão de projeto Façade⁸ e faz uso de uma classe não pública chamada `DiscoveryAdapter`, que provê uma forma simplificada de implementar a interface `javax.bluetooth.DiscoveryListener`.

No pacote `br.com.staroski.obdjrj.http` encontra-se a classe `Http`, que também é uma implementação do padrão de projeto Façade e provê uma interface simplificada para realizar requisições HTTP GET e HTTP POST. A classe `Http` realiza estas requisições respectivamente através de duas classes não públicas chamadas `GetRequest` e `PostRequest`.

Na Figura 19 é apresentado o digrama de pacotes do firmware e a forma como se relacionam as classes mais relevantes.

Figura 19 - Relacionamento entre as principais classes do firmware



Fonte: Elaborado pelo autor.

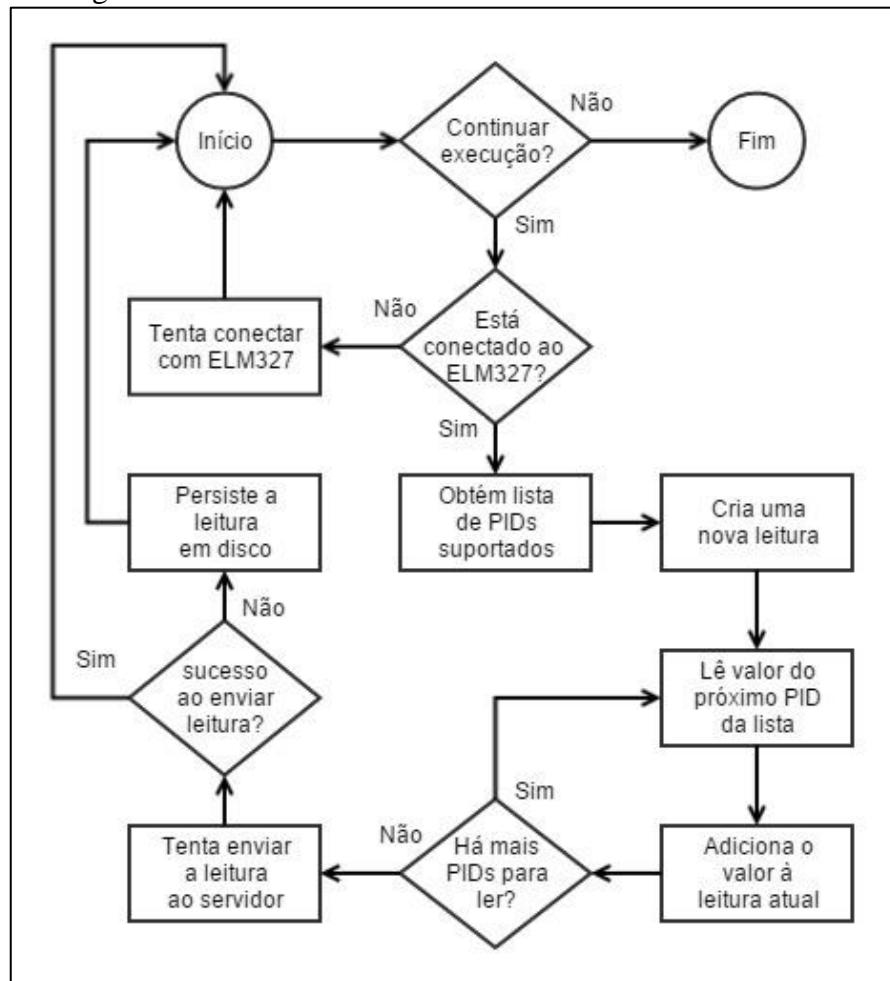
⁸ Rocha (2005) explica que o padrão Façade define uma interface de nível mais elevado que torna o subsistema mais fácil de usar.

As classes `Scanner` e `DataMonitor`, apresentadas na Figura 19, correspondem aos dois processos executados pelo firmware. A classe `Scanner` é composta de um atributo do tipo `ELM327`, responsável pela comunicação com a interface ELM327. A classe `ELM327` é composta de um atributo do tipo `IO`, que abstrai o meio de comunicação utilizado (Socket ou Bluetooth). Tanto a classe `Scanner` quanto a classe `DataMonitor` utilizam a classe `Config` para obter as configurações definidas no arquivo `obd-jrp.properties`. A classe `Scanner` permite que outras classes se registrem a ela através da interface `ScannerListener`, de forma a serem notificadas quando ocorre um ciclo de leitura ou um erro. As notificações de leitura são propagadas na forma de objetos do tipo `Scan`, que agregam objetos do tipo `Data`, correspondentes aos PIDs lidos pela interface ELM327. Objetos do tipo `Data` encapsulam um PID e seu respectivo valor na forma de bytes, para transformar estes bytes informação humanamente legível, é necessário utilizar a classe `Parsing`. A classe `Parsing` submete um objeto `Data` à uma implementação da interface `Parser`, capaz de transformar um objeto do tipo `Data` em um objeto do tipo `Parsed`, que possui informação humanamente legível. A classe `DataMonitor` adiciona em um objeto do tipo `Package` os objetos `Scan` que não foram enviados ao servidor, para posteriormente serem enviados em uma única requisição.

3.2.1.1 LEITURA DE DADOS DA INTERFACE ELM327 BLUETOOTH

Conforme citado na seção 3.2, o ciclo de vida do firmware consiste na execução de dois processos paralelos. O primeiro processo especificado corresponde à leitura dos dados da interface ELM327 Bluetooth. Esta leitura é realizada em um laço, apresentado na Figura 20.

Figura 20 - Leitura de dados da interface ELM327 Bluetooth



Fonte: Elaborado pelo autor.

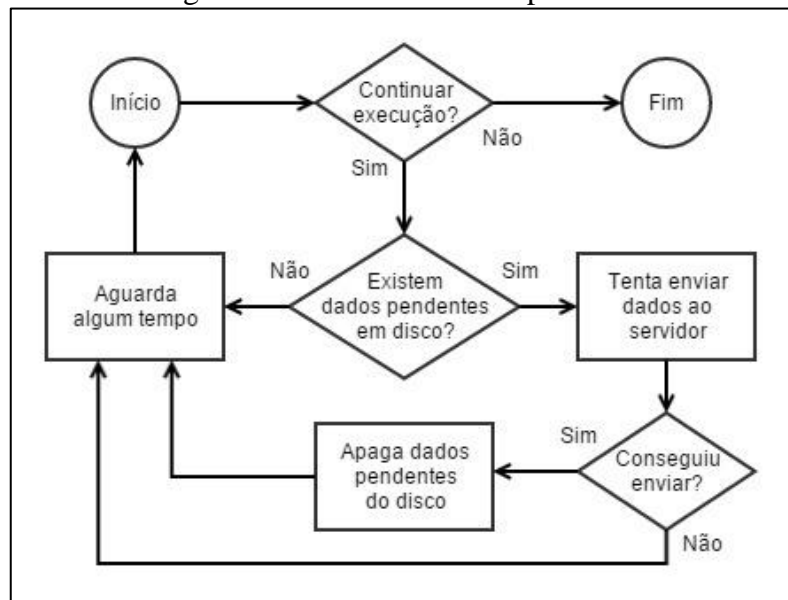
Na Figura 20 é apresentado o algoritmo de leitura da interface ELM327 Bluetooth, que consiste em um laço de repetição que realiza as seguintes operações:

- a) se estiver conectado ao ELM327, executa o passo c), senão executa o passo b);
- b) tenta conectar-se ao ELM327 e volta ao passo a);
- c) solicita ao ELM327 a lista dos PIDs suportados pelo veículo e executa o passo d);
- d) cria um objeto de leitura para armazenar os valores dos PIDs e executa o passo e);
- e) obtém o valor do próximo PID suportado e executa o passo f);
- f) adiciona o valor do PID lido ao objeto de leitura e executa o passo g);
- g) se houver mais PIDs para ler, volta ao passo e), senão executa o passo h);
- h) tenta enviar o objeto de leitura ao servidor através de uma requisição HTTP GET e executa o passo i);
- i) se conseguiu enviar o objeto de leitura, volta ao passo a), senão executa o passo j);
- j) persiste o objeto de leitura em disco e volta ao passo a).

3.2.1.2 ENVIO DOS DADOS PENDENTES

O segundo processo do ciclo de vida do firmware é o envio dos dados pendentes. Para este processo funcionar, é necessário que as configurações de rede do Raspberry Pi estejam possibilitando o acesso à internet. Não existe restrição quanto ao meio acesso, pode ser linha discada, DSL, 3G, WiFi, cabo ou outros. O processo de envio de dados trata de monitorar o diretório onde o processo de leitura dos dados persistiu as leituras que não puderam ser enviadas ao servidor. As leituras são empacotadas em um único arquivo e enviadas ao servidor através de uma requisição HTTP POST. Na Figura 21 observa-se o fluxo do processo de transmissão dos dados pendentes.

Figura 21 - Envio dos dados pendentes



Fonte: Elaborado pelo autor.

Assim como a leitura dos dados da interface ELM327 Bluetooth, o envio dos dados pendentes também consiste em um laço de repetição que realiza as seguintes operações:

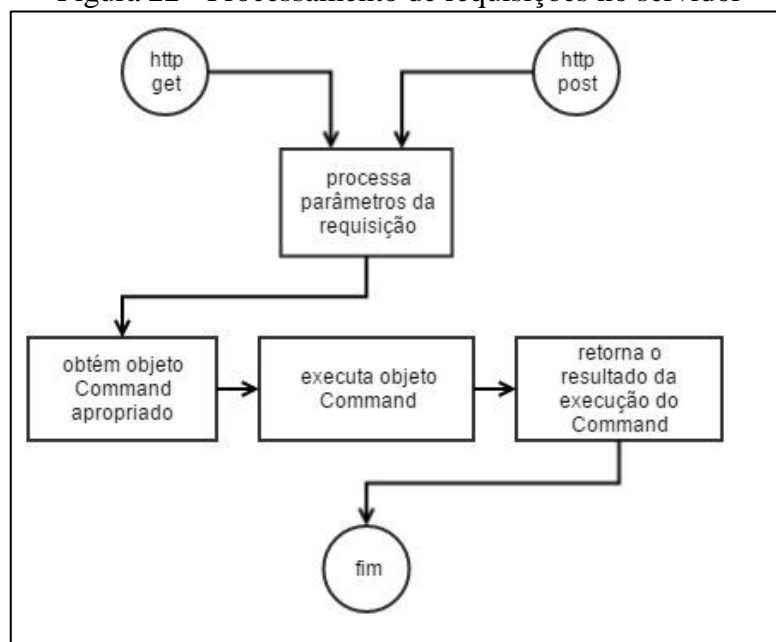
- verifica se há arquivos de dados pendentes no diretório, se houver, executa o passo c), senão executa o passo b);
- aguarda 5 minutos e volta ao passo a);
- tenta enviar cada arquivo de dados pendentes ao servidor através de uma requisição HTTP POST e segue ao passo d);
- apaga do disco, cada arquivo enviado com sucesso ao servidor, e volta ao passo b).

3.2.2 ESPECIFICAÇÃO DO SERVIDOR

O desenvolvimento do servidor foi realizado em uma única camada, que tem como ponto de entrada um Servlet Java EE, capaz de processar tanto requisições HTTP GET quanto

requisições HTTP POST. Para processar as requisições, utilizou-se o padrão de projeto Command⁹, de forma que, a partir dos parâmetros recebidos, se obtenha um objeto apropriado para tratar a requisição. Na Figura 22 observa-se como ocorre o fluxo de processamento de requisições no servidor.

Figura 22 - Processamento de requisições no servidor



Fonte: Elaborado pelo autor.

O pacote que define as classes do servidor é chamado `br.com.staroski.obdjrp.web`, nele se encontram as seguintes classes:

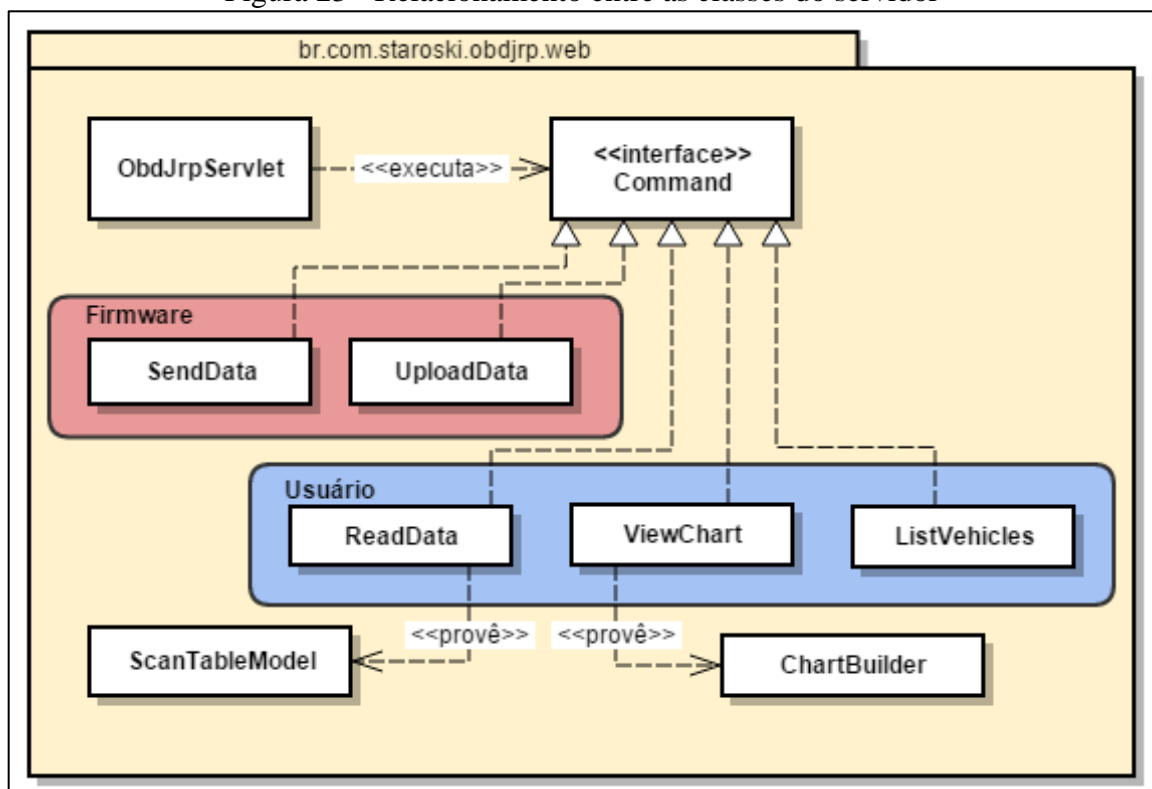
- a) `ObdJrpServlet`, é o ponto de entrada do servidor. Esta classe intercepta as requisições HTTP GET e HTTP POST, delegando a execução para uma implementação apropriada de `Command`;
- b) `Command`, interface para os objetos que tratam parâmetros específicos das requisições recebidas pelo `ObdJrpServlet`;
- c) `SendData`, implementação de `Command` invocada pelo firmware para enviar leituras em tempo real;
- d) `UploadData`, implementação de `Command` invocada pelo firmware para enviar pacotes de leituras pendentes;
- e) `ListVehicles`, implementação de `Command` invocada pelo usuário ao abrir a página com a lista de veículos;

⁹ Segundo Rocha (2005), o padrão Command encapsula uma requisição na forma de um objeto, permitindo que clientes parametrizem diferentes requisições.

- f) `ReadData`, implementação de `Command` invocada pelo usuário ao abrir a página para visualizar leituras em tempo real;
- g) `ViewChart`, implementação de `Command` invocada pelo usuário ao selecionar uma leitura para visualização gráfica;
- h) `ScanTableModel`, provê métodos para a página renderizar uma tabela com as leituras;
- i) `ChartBuilder`, provê métodos para a página renderizar os gráficos das leituras.

Na Figura 23 é apresentado o diagrama de classes reduzido do servidor. Observa-se que algumas classes estão delimitadas em uma região vermelha e outras em uma região azul. Estas regiões respectivamente representam os recursos utilizados pelo firmware e os recursos utilizados pelo usuário ao acessar as páginas através de um navegador web.

Figura 23 - Relacionamento entre as classes do servidor



Fonte: Elaborado pelo autor.

As classes `SendData` e `UploadData`, apresentadas na Figura 23, correspondem aos recursos acessados pelo firmware para enviar dados em tempo real e enviar pacotes de dados pendentes respectivamente. As classes `ReadData`, `ViewChart`, `ListVehicles`, `ScanTableModel` e `ChartBuilder` correspondem aos recursos utilizados pelo usuário ao acessar nas páginas web do servidor.

3.3 IMPLEMENTAÇÃO

Nesta seção são apresentados os aspectos sobre a preparação do ambiente de execução no Raspberry Pi, as implementações do firmware, do servidor, as ferramentas e técnicas utilizadas para a construção do protótipo.

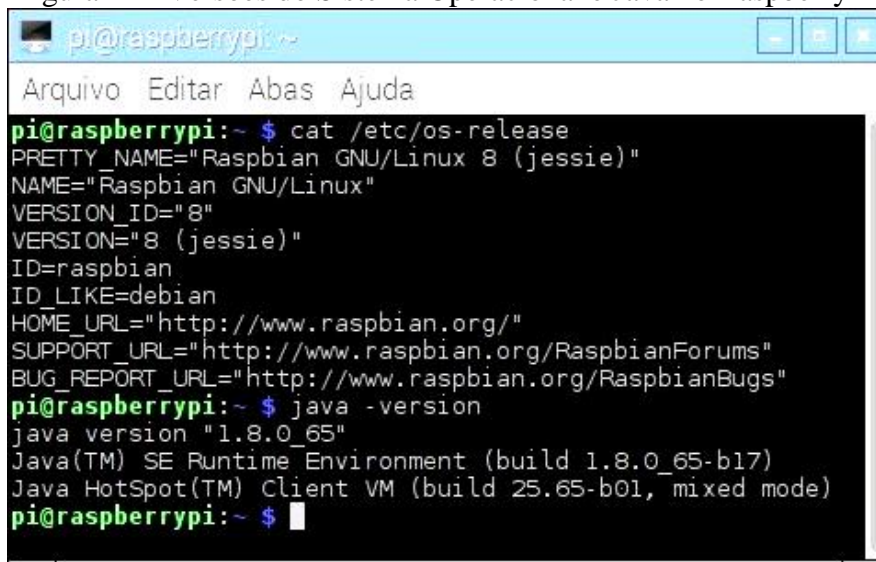
3.3.1 Técnicas e ferramentas utilizadas

As implementações tanto do firmware quanto do servidor, foram realizadas utilizando o ambiente de desenvolvimento Eclipse Neon com linguagem de programação Java. Para o desenvolvimento do firmware foi utilizado a Application Program Interface (API) do Java SE e a API BlueCove para realizar a comunicação com a interface ELM327 Bluetooth. Para o desenvolvimento do servidor foi utilizado a API do Java SE, Java EE e API Google Charts, para criar gráficos em linguagem JavaScript. Os diagramas foram elaborados através da ferramenta Gliffy Online.

3.3.1.1 Preparação do ambiente de execução no Raspberry Pi 3 Model B

O sistema operacional instalado no Raspberry Pi é o Raspbian GNU/Linux 8, que é disponibilizada com a versão 1.8 do Java (como pode ser observado no terminal apresentado na Figura 24).

Figura 24 - Versões do Sistema Operacional e Java no Raspberry Pi

A terminal window titled 'pi@raspberrypi: ~' with a menu bar containing 'Arquivo', 'Editar', 'Abas', and 'Ajuda'. The terminal shows the output of two commands: 'cat /etc/os-release' and 'java -version'. The first command outputs details about Raspbian GNU/Linux 8 (jessie), including its name, version ID, and various URLs. The second command outputs the Java version as 1.8.0_65, identifying it as the SE Runtime Environment and HotSpot(TM) Client VM.

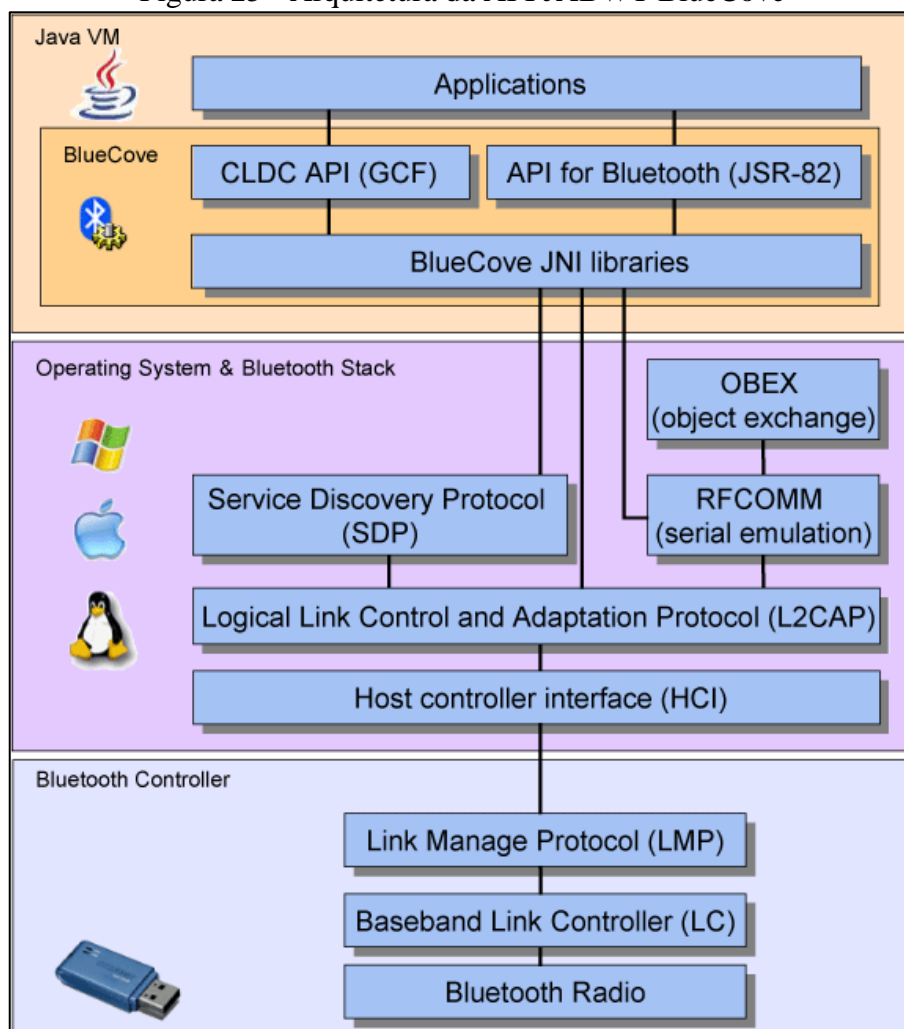
```
pi@raspberrypi:~ $ cat /etc/os-release
PRETTY_NAME="Raspbian GNU/Linux 8 (jessie)"
NAME="Raspbian GNU/Linux"
VERSION_ID="8"
VERSION="8 (jessie)"
ID=raspbian
ID_LIKE=debian
HOME_URL="http://www.raspbian.org/"
SUPPORT_URL="http://www.raspbian.org/RaspbianForums"
BUG_REPORT_URL="http://www.raspbian.org/RaspbianBugs"
pi@raspberrypi:~ $ java -version
java version "1.8.0_65"
Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
Java HotSpot(TM) Client VM (build 25.65-b01, mixed mode)
pi@raspberrypi:~ $
```

Fonte: Elaborado pelo autor.

Conforme citado na seção 2.5, o Raspberry Pi funciona de forma análoga à um PC e, para realizar a comunicação via Bluetooth com Java, foi utilizado a biblioteca BlueCove, que não faz parte da distribuição padrão do Java. BlueCove é uma Java API for Bluetooth Wireless Technology (JABWT), que implementa a Java Specification Request 82 (JSR-82)

(BLUECOVE, 2008, tradução nossa). A biblioteca BlueCove utiliza Java Native Interface (JNI) para a comunicação via Bluetooth, trocando mensagens diretamente com os drivers do sistema operacional (como pode ser observado na arquitetura apresentada na Figura 25).

Figura 25 - Arquitetura da API JABWT BlueCove



Fonte: BlueCove (2008).

Observa-se que no repositório¹⁰ da biblioteca BlueCove estão disponíveis versões compiladas para arquitetura x86, entretanto o Raspberry Pi possui arquitetura ARM, sendo necessário recompilar o código fonte no próprio dispositivo. Alderton (2015) explica que, para compilar o código fonte da biblioteca BlueCove no Raspberry Pi 2, é necessário instalar os pacotes `bluetooth`, `bluez-utils` e `blueman`. Este procedimento não funcionou no Raspberry Pi 3 Model B, somente o pacote `bluetooth` foi instalado com sucesso, já os pacotes `bluez-utils` e `blueman` não são suportados. Somente foi possível compilar o código fonte da biblioteca BlueCove após instalar os pacotes `libbluetooth-dev`, `bluez`, `bluez-cups` e

¹⁰ Repositório do BlueCove: <http://repo1.maven.org/maven2/net/sf/bluecove/bluecove>.

bluez-obexd no Raspberry Pi. O processo de compilação do código fonte está detalhado nos arquivos `read-me.txt` e `developer-read-me.txt`, disponíveis no repositório da biblioteca BlueCove. Com base nas instruções destes arquivos, foi necessário:

- a) instalar as ferramentas Maven e Ant no Raspberry Pi;
- b) executar o comando Maven para criar os diretórios de código fonte compatíveis com o ambiente Eclipse:


```
- mvn eclipse:clean eclipse:eclipse -DdownloadSources=true;
```
- c) executar o comando Ant para compilar as bibliotecas nativas e gerar o arquivo JAR da biblioteca Java:


```
- ant all.
```

Com a biblioteca BlueCove compilada para arquitetura ARM, foi possível realizar no próprio Raspberry algumas provas de conceito para avaliar se seria viável dar continuidade ao desenvolvimento utilizando a linguagem Java. Os testes consistiram em listar os dispositivos Bluetooth pareados e tentar obter a lista de serviços Bluetooth disponíveis. Como os testes foram positivos, deu-se continuidade ao desenvolvimento do protótipo utilizando a linguagem Java. Caso os testes com o BlueCove não fossem positivos, uma alternativa seria pesquisar bibliotecas para comunicação Bluetooth da linguagem Python¹¹, que também é disponibilizada com o Raspian GNU/Linux 8.

De forma a executar os programas automaticamente quando o Raspberry Pi for ligado, foi necessário editar o arquivo `.config/lxsession/LXDE-pi/autostart` incluindo as linhas `@lxterminal --working-directory=/home/pi --command="/home/pi/obdjrp-upload"` e `@lxterminal --working-directory=/home/pi --command="/home/pi/obdjrp-scan"`, onde `obdjrp-upload` é um script que executa a classe `ObdJrpUploadData` e `obdjrp-scan` é um script que executa a classe `ObdJrpScanData`. As classes `ObdJrpUploadData` e `ObdJrpScanData` serão citadas posteriormente. Esta configuração é responsável pela autonomia do protótipo, pois ao ligar o Raspberry Pi, os programas serão executados sem necessidade de interação humana.

3.3.1.2 Listando dispositivos Bluetooth no firmware

A primeira aplicação desenvolvida para o firmware foi um programa em linha de comando, chamado `ObdJrpListDevices`, que lista os dispositivos Bluetooth pareados e os

¹¹ A API do trabalho correlato PyOBD poderia ser utilizado para o desenvolvimento em Python.

serviços Bluetooth disponíveis. Este programa serviu como prova de conceito para a viabilidade de utilizar a linguagem Java para comunicação Bluetooth no Raspberry Pi. No Quadro 1 é apresentado o código fonte da classe `ObdJrpListDevices`, que procura obter a lista de dispositivos Bluetooth pareados e identificar os serviços que os mesmos disponibilizam.

Quadro 1 - Listando dispositivos e serviços Bluetooth

```

11 public final class ObdJrpListDevices extends ObdJrpApp {
12
13     public static void main(String[] args) {
14         try {
15             ObdJrpListDevices program = new ObdJrpListDevices();
16             program.execute();
17         } catch (Throwable t) {
18             t.printStackTrace();
19             System.exit(-1);
20         }
21     }
22
23     private ObdJrpListDevices() throws IOException {
24         super("list-devices");
25     }
26
27     private void execute() throws IOException {
28         List<RemoteDevice> devices = Bluetooth.listDevices();
29         for (RemoteDevice device : devices) {
30             String address = device.getBluetoothAddress();
31             String name = device.getFriendlyName(false);
32             System.out.printf("device \"%s\" - \"%s\" ", address, name);
33             printServices(device);
34             System.out.printf("\n\n");
35         }
36     }
37
38     private void printServices(RemoteDevice device) throws IOException {
39         List<ServiceRecord> services = Bluetooth.listServices(device);
40         for (ServiceRecord service : services) {
41             System.out.printf("%n\t\"%s\"%n", Bluetooth.getServiceName(service));
42         }
43     }
44 }

```

Fonte: Elaborado pelo autor.

Observa-se na linha 28 da classe `ObdJrpListDevices`, a invocação do método estático `Bluetooth.listDevices`. A classe `Bluetooth` foi desenvolvida para abstrair a complexidade da JABWT implementada pela biblioteca `BlueCove`. Thompson, Kline e Kumar (2008, p. 136), explicam que antes de consultar um dispositivo, é necessário que a aplicação implemente a interface `DiscoveryListener` e os métodos `deviceDiscovered` e `inquiryCompleted`.

Para realizar a consulta de dispositivos, é necessário invocar o método `startInquiry` da classe `DiscoveryAgent`, passando por parâmetro a instância do `DiscoveryListener`. No Quadro 2 é apresentado o código fonte do método `Bluetooth.listDevices`. Na linha 146 é invocado o método `startInquiry`, passando como parâmetro o objeto `DEVICE_LISTENER`, este objeto é uma instância de `DiscoveryListener`.

Quadro 2 - Disparando consulta de dispositivos com JABWT

```

142 public static List<RemoteDevice> listDevices() throws IOException {
143     synchronized (LOCK) {
144         DiscoveryAgent agent = LocalDevice.getLocalDevice().getDiscoveryAgent();
145         agent.cancelInquiry(DEVICE_LISTENER);
146         DEVICE_LISTENER.reset();
147         if (agent.startInquiry(DiscoveryAgent.GIAC, DEVICE_LISTENER)) {
148             LOCK.lock();
149         }
150     }
151     return DEVICE_LISTENER.getDevices();
152 }

```

Fonte: Elaborado pelo autor.

Observa-se na linha 147 que, caso a invocação do `startInquiry` retorne `true`, é invocado o método `LOCK.lock`. Este método faz com que a execução do método `Bluetooth.listDevices` permaneça bloqueada até que outra Thread invoque o método `LOCK.unlock`. Esse bloqueio é necessário pois o método `startInquiry` retorna imediatamente após ser invocado. Esse retorno imediato ocorre pois a consulta de dispositivos ocorre concorrentemente em outra Thread disparada pelo `startInquiry`, o que justifica a necessidade de registrar-se uma instância de `DiscoveryListener` para ser notificada pelo método `deviceDiscovered` quando um dispositivo é encontrado e pelo método `inquiryCompleted` quando a consulta termina. A implementação do `DiscoveryListener` utilizado é apresentada no Quadro 3.

Quadro 3 - DiscoveryListener para consulta de dispositivos

```

21 // listener para descoberta de dispositivos
22 private static final DeviceDiscovery DEVICE_LISTENER = new DeviceDiscovery();
23
24 private static class DeviceDiscovery extends DiscoveryAdapter {
25
26     private final List<RemoteDevice> devices = new LinkedList<>();
27
28     @Override
29     public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {
30         devices.add(btDevice);
31     }
32
33     public List<RemoteDevice> getDevices() {
34         return devices;
35     }
36
37     @Override
38     public void inquiryCompleted(int discType) {
39         LOCK.unlock();
40     }
41
42     public void reset() {
43         devices.clear();
44     }
45 }

```

Fonte: Elaborado pelo autor.

Na linha 30 do `DiscoveryListener`, o dispositivo recebido pelo parâmetro `btDevice` é adicionado à lista `devices` e na linha 39, quando a consulta termina, é invocado o método `LOCK.unlock`, de forma a desbloquear o método `Bluetooth.listDevices` e retornar a lista de dispositivos descobertos.

Optou-se em escrever uma classe alternativa para sincronização de processos, pois os métodos `java.lang.Object.wait`, `java.util.concurrent.locks.Lock.lock` e `java.util.concurrent.Semaphore.acquire`, declaram o lançamento da exceção checada `InterruptedException`, forçando o desenvolvedor a tratar ou relançar a exceção. Os métodos da classe `Lock` criada, não declaram o lançamento de nenhuma exceção checada, e o método `lock` trata a `InterruptedException` e a transforma em uma exceção não checada do tipo `RuntimeException`. Essa técnica torna mais limpo o código onde o método `lock` for utilizado. No Quadro 4 observa-se a implementação da classe `Lock`.

Quadro 4 - Classe `Lock` utilizada para sincronização de processos

```

3 public final class Lock {
4
5     private final Object LOCK = new Object();
6
7     public void lock() {
8         lock(0);
9     }
10
11     public void lock(long timeout) {
12         synchronized (LOCK) {
13             try {
14                 LOCK.wait(timeout);
15             } catch (InterruptedException e) {
16                 throw new RuntimeException("Lock interrupted!", e);
17             }
18         }
19     }
20
21     public void unlock() {
22         synchronized (LOCK) {
23             LOCK.notifyAll();
24         }
25     }
26 }

```

Fonte: Elaborado pelo autor.

O processo de consulta de serviços utilizando JABWT é idêntico ao processo de lista consulta de dispositivos. É necessário invocar o método `searchServices`, mas além de passar por parâmetro um objeto do tipo `DiscoveryListener`, também é necessário passar o objeto `RemoteDevice` do qual se deseja obter os serviços disponíveis, um array com os atributos que se deseja obter dos serviços e um array de Universally Unique Identifier (UUID) correspondente ao perfil de serviço Bluetooth que se deseja listar. No trabalho desenvolvido o

único atributo de interesse é o nome do serviço e o perfil Serial Port Profile (SPP). Segundo Bluetooth (2014, p. 1949), o valor hexadecimal correspondente ao atributo `ServiceName` é `0x0100` e segundo a documentação¹² da classe `javax.bluetooth.UUID`, o valor hexadecimal correspondente ao perfil SPP é `0x1101`. No Quadro 5 é apresentado a implementação do método `Bluetooth.listServices`, que realiza a consulta dos serviços disponíveis para o dispositivo informado no parâmetro `device`.

Quadro 5 - Disparando consulta de serviços com JABWT

```

154 public static List<ServiceRecord> listServices(RemoteDevice device) throws IOException {
155     synchronized (LOCK) {
156         int[] attributes = new int[] { Bluetooth.NAME };
157         UUID[] uuids = new UUID[] { Bluetooth.SPP };
158         DiscoveryAgent agent = LocalDevice.getLocalDevice().getDiscoveryAgent();
159         agent.cancelServiceSearch(SERVICE_LISTENER.getTransactionID());
160         SERVICE_LISTENER.reset();
161         int id = agent.searchServices(attributes, uuids, device, SERVICE_LISTENER);
162         SERVICE_LISTENER.setTransactionID(id);
163         if (id > 0) {
164             LOCK.lock();
165         }
166     }
167     return SERVICE_LISTENER.getServices();
168 }

```

Fonte: Elaborado pelo autor.

Observa-se entre as linhas 161 e 163 que caso a invocação do `searchServices`, retorne um `id` maior que zero, é invocado o método `LOCK.lock`, fazendo com que a execução do método `Bluetooth.listServices` permaneça bloqueada até que outra Thread invoque o método `LOCK.unlock`. Esse bloqueio é necessário pois o método `searchServices` também é assíncrono e retorna imediatamente após ser invocado. Também se faz necessário registrar uma instância de `DiscoveryListener` para ser notificada pelo método `servicesDiscovered` quando serviços são descobertos e pelo método `serviceSearchCompleted` quando a consulta termina. A implementação do `DiscoveryListener` utilizado para a descoberta de serviços é apresentada no Quadro 6.

¹² Documentação da classe `UUID`: <http://www.bluecove.org/bluecove/apidocs/javax/bluetooth/UUID.html>

Quadro 6 - DiscoveryListener para consulta de serviços

```

47 // listener para descoberta de servicos
48 private static final ServiceDiscovery SERVICE_LISTENER = new ServiceDiscovery();
49
50 private static class ServiceDiscovery extends DiscoveryAdapter {
51
52     private final List<ServiceRecord> services = new LinkedList<>();
53
54     private int transactionID;
55
56     public List<ServiceRecord> getServices() {
57         return services;
58     }
59
60     public int getTransactionID() {
61         return transactionID;
62     }
63
64     public void reset() {
65         services.clear();
66     }
67
68     @Override
69     public void servicesDiscovered(int transID, ServiceRecord[] servRecord) {
70         for (ServiceRecord service : servRecord) {
71             services.add(service);
72         }
73     }
74
75     @Override
76     public void serviceSearchCompleted(int transID, int respCode) {
77         LOCK.unlock();
78     }
79
80     public void setTransactionID(int transactionID) {
81         this.transactionID = transactionID;
82     }
83 }

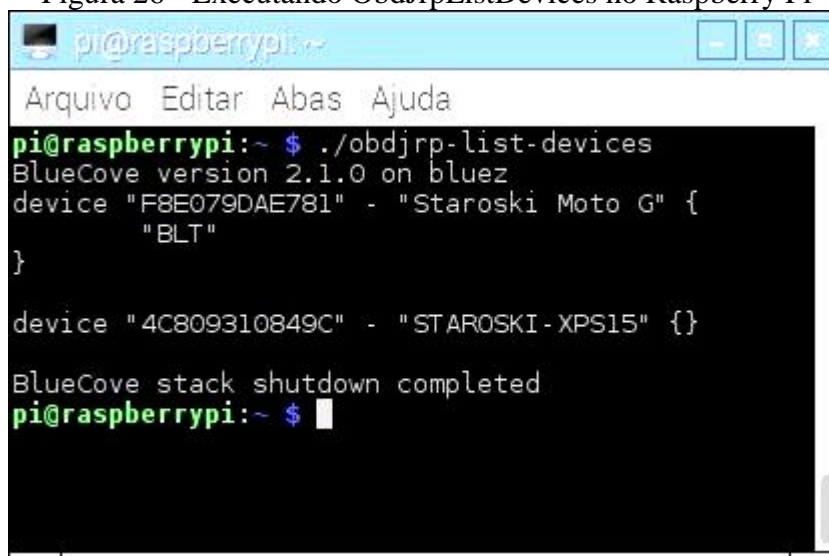
```

Fonte: Elaborado pelo autor.

Na linha 71 do `DiscoveryListener`, cada serviço descoberto recebido pelo parâmetro `servRecord` é adicionado à lista `services` e na linha 77, quando a consulta termina, é invocado o método `LOCK.unlock`, de forma a desbloquear o método `Bluetooth.listServices` e retornar a lista de serviços descobertos para o dispositivo informado.

Na Figura 26 observa-se o resultado da execução da classe `ObdJrpListDevices` no Raspberry Pi, listando dois dispositivos. O primeiro dispositivo possui o endereço `F8E079DAE781`, nome "Staroski Moto G" e disponibiliza um serviço chamado "BLT". O segundo dispositivo possui endereço `4C809310849C`, nome "STAROSKI-XPS15" e não possui nenhum serviço disponível.

Figura 26 - Executando ObdJrpListDevices no Raspberry Pi



```

pi@raspberrypi:~
Arquivo  Editar  Abas  Ajuda
pi@raspberrypi:~ $ ./obdjrp-list-devices
BlueCove version 2.1.0 on bluez
device "F8E079DAE781" - "Staroski Moto G" {
    "BLT"
}

device "4C809310849C" - "STAROSKI-XPS15" {}

BlueCove stack shutdown completed
pi@raspberrypi:~ $

```

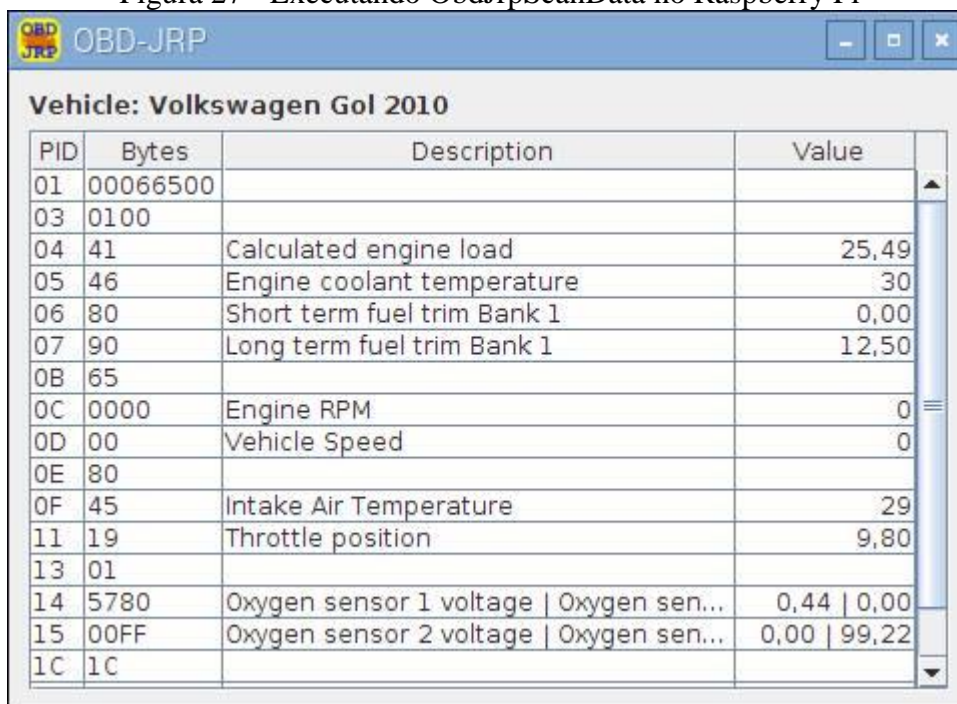
Fonte: Elaborado pelo autor.

3.3.1.3 Leitura de dados em tempo real

A segunda aplicação desenvolvida para o firmware foi um programa em linha de comando chamado `ObdJrpScanData`, que se comunica via Bluetooth com a interface ELM327 conectada à porta OBD2 do veículo. Embora o `ObdJrpScanData` seja um programa em linha de comando, ele apresenta uma interface gráfica contendo uma lista com os PIDs e respectivos valores lidos em tempo real. Esta janela foi desenvolvida pois tentou-se conectar um display touchscreen de 3,5 polegadas ao Raspberry Pi, de forma que as leituras dos PIDs pudessem ser observadas dentro do próprio veículo. Entretanto o display não apresentou compatibilidade com o Raspberry Pi 3. Mesmo assim, optou-se em manter a janela no programa de forma a auxiliar nos testes realizados. Para visualizar a área de trabalho do Raspberry Pi utilizou-se um monitor com entrada HDMI e para acessar sua área de trabalho enquanto conectado ao veículo, utilizou-se o acesso remoto através do VNC¹³. Na Figura 27 observa-se o aspecto da janela apresentada pelo `ObdJrpScanData`.

¹³ VNC é um software de acesso remoto, disponível em <https://www.realvnc.com/raspberrypi>, que é disponibilizado gratuitamente com o sistema Raspbian, para fins não comerciais.

Figura 27 - Executando ObdJrpScanData no Raspberry Pi



PID	Bytes	Description	Value
01	00066500		
03	0100		
04	41	Calculated engine load	25,49
05	46	Engine coolant temperature	30
06	80	Short term fuel trim Bank 1	0,00
07	90	Long term fuel trim Bank 1	12,50
0B	65		
0C	0000	Engine RPM	0
0D	00	Vehicle Speed	0
0E	80		
0F	45	Intake Air Temperature	29
11	19	Throttle position	9,80
13	01		
14	5780	Oxygen sensor 1 voltage Oxygen sen...	0,44 0,00
15	00FF	Oxygen sensor 2 voltage Oxygen sen...	0,00 99,22
1C	1C		

Fonte: Elaborado pelo autor.

Para se comunicar com a interface Bluetooth, o `ObdJrpScanData` precisa conhecer o endereço Bluetooth do ELM327 e o nome do serviço disponibilizado. Estas informações são configuradas em um arquivo chamado `obd-jrp.properties` que é representado pela classe `Config`. Para obter o endereço Bluetooth do dispositivo e o nome do serviço disponibilizado. Para descobrir o endereço do dispositivo e o nome do serviço, executou-se a classe `ObdJrpListDevices` enquanto a interface ELM327 estava conectada à um veículo. Com as informações obtidas, editou-se o arquivo `obd-jrp.properties` com as propriedades apresentadas no Quadro 7.

Quadro 7 - Configuração de acesso Bluetooth

```
1# connection_type tem que ser bluetooth ou socket
2# se for bluetooth, tem que informar
3# bluetooth_device e bluetooth_service
4# se for socket, tem que informar
5# socket_address e socket_port
6 connection_type=bluetooth
7
8# adaptador bluetooth
9 bluetooth_device=000000000001
10 bluetooth_service=SPP
11
12# adaptador wifi
13 socket_address=192.168.0.10
14 socket_port=35000
```

Fonte: Elaborado pelo autor.

A primeira propriedade definida no arquivo `obd-jrp.properties` chama-se `connection_type`. Esta propriedade determina qual o tipo de conexão a ser utilizada. A implementação da classe `Config` só interpreta dois valores para o `connection_type`: `bluetooth` e `socket`. O suporte para conexões via `Socket` foi implementado pois, enquanto não foi possível compilar a biblioteca `BlueCove` no Raspberry Pi, utilizou-se uma interface ELM327 WiFi. Entretanto tal abordagem se demonstrou inviável pois quando as configurações de rede do Raspberry Pi são modificadas para acessar o ponto de rede WiFi da interface ELM327, o dispositivo fica impossibilitado de conectar-se à internet. Na Figura 28 é apresentado o aspecto das duas interfaces utilizadas durante o desenvolvimento do protótipo. Percebe-se que, embora sejam acessíveis por meios diferentes, seu encapsulamento é semelhante.

Figura 28 - Interfaces ELM327 WiFi e Bluetooth



Fonte: Elaborado pelo autor.

Com as configurações de acesso definidas no arquivo `obd-jrp.properties`, a classe `ObdJrpScanData` obtém, através da classe `Config`, uma instância da interface `IO`. Independentemente do meio de acesso ser `Bluetooth` ou `Socket`, a classe `Config` sempre disponibiliza um objeto do tipo `IO` que abstrai o meio de comunicação. O objeto `IO` é parâmetro de construção para objetos do tipo `Scanner`. Após instanciar um `Scanner`, pode-se registrar objetos do tipo `ScannerListener`, que serão notificados quando um ciclo de leitura for concluído ou quando ocorre um erro na comunicação com a interface ELM327. Para finalmente inicializar a leitura, é invocado o método `start` do objeto `Scanner`. Este procedimento é executado pelo programa `ObdJrpScanData` na implementação do método `execute`, como pode ser visto no Quadro 8.

Quadro 8 - Método execute da classe ObdJrpScanData

```

47 private void execute() {
48     final IO connection = Config.get().connection();
49     final ScannerWindow window = getScannerWindow();
50     final ScannerListener windowListener = window.getObdJrpListener();
51     window.setVisible(true);
52     while (!connection.isOpen()) {
53         System.out.println("trying to connect with " + connection);
54         try {
55             scanner = new Scanner(connection.open());
56             scanner.addListener(errorListener);
57             scanner.addListener(windowListener);
58             scanner.start();
59             System.out.println("successfull connected!");
60         } catch (IOException | ELM327Error error) {
61             Print.message(error);
62             if (scanner != null) {
63                 scanner.removeListener(errorListener);
64                 scanner.removeListener(windowListener);
65             }
66             connection.close();
67         }
68     }
69 }

```

Fonte: Elaborado pelo autor.

Observa-se nas linhas 56 e 57 que a classe ObdJrpScanData registra dois objetos que implementam a interface ScannerListener:

- a) errorListener, responsável por tentar reestabelecer a conexão com a interface ELM327 caso ocorra um erro. Sua implementação é apresentada no Quadro 9;
- b) windowListener, responsável por atualizar a janela que apresenta as leituras em tempo real. Sua implementação é apresentada no Quadro 11.

Quadro 9 - Reestabelecendo conexão Bluetooth após erro

```

33 private final ScannerListener errorListener = new ScannerListener() {
34
35     @Override
36     public void onError(ELM327Error error) {
37         restartAfterError(error);
38     }
39
40     @Override
41     public void onScanned(Scan scannedData) { /* ignora */ }
42 };

```

Fonte: Elaborado pelo autor.

Observa-se que na linha 41 é ignorado o evento de leitura recebida, pois o ScannerListener apenas trata o evento de erro, na linha 37 e executa o método restartAfterError, responsável por interromper o objeto Scanner através do método stop e tentar se reconectar através do método execute. A implementação dos métodos restartAfterError e stop é apresentada no Quadro 10.

Quadro 10 - Métodos restartAfterError e stop da classe ObdJrpScanData

```

96 private void restartAfterError(final ELM327Error error) {
97     new Thread(new Runnable() {
98
99         @Override
100         public void run() {
101             stop();
102             execute();
103         }
104     }, "Scanner_Restarter").start();
105 }
106
107 private void stop() {
108     if (scanner != null) {
109         scanner.stop();
110         System.out.println("stopped scanning!");
111     }
112 }

```

Fonte: Elaborado pelo autor.

Na linha 104 é disparada uma nova Thread que irá executar os métodos `stop` e `execute`, que podem ter uma execução demorada. Como o método `restartAfterError` foi invocado a partir de um tratamento de evento, é disparada uma Thread para que o tratador de evento não permaneça bloqueado aguardando o fim da execução do `stop` e `execute`.

Quadro 11 - Atualizando interface de usuário em tempo real

```

118 @Override
119 public void onError(ELM327Error error) {
120     labelVehicle.setText(Print.message(error));
121 }
122
123 @Override
124 public void onScanned(Scan scannedData) {
125     dataList = scannedData.getData();
126     OBD2DataModel model = (OBD2DataModel) table.getModel();
127     model.update();
128 }
129 }

```

Fonte: Elaborado pelo autor.

Na linha 125 é recebido o parâmetro `scannedData`, um objeto do tipo `Scan`, que contém as leituras obtidas pela interface ELM327. Esse parâmetro é atribuído à variável de instância `dataList` e em seguida, nas linhas 126 e 127 é realizada a atualização de um objeto do tipo `javax.swing.table.TableModel`¹⁴ no qual parte relevante da implementação é apresentada no Quadro 12.

¹⁴ Documentação disponível em <https://docs.oracle.com/javase/8/docs/api/javax/swing/table/TableModel.html>.

Quadro 12 - Apresentando dados lidos na interface de usuário

```

65 @Override
66 public Object getValueAt(int row, int col) {
67     Data rawData = dataList.get(row);
68     Parsed translated = Parsing.parse(rawData);
69     switch (col) {
70         case 0:
71             return rawData.getPID();
72         case 1:
73             return rawData.getValue();
74         case 2:
75             return translated.getDescriptions(" | ");
76         case 3:
77             default:
78                 return translated.getValues(" | ");
79     }
80 }

```

Fonte: Elaborado pelo autor.

Na linha 66 observa-se que o método `getValueAt` recebe dois parâmetros, `row` e `col`, que correspondem aos índices da linha e coluna da célula a ser renderizada pelo componente visual `javax.swing.JTable`¹⁵. Na linha 67 utiliza-se o parâmetro `row`, para acessar um elemento `Data` do objeto `dataList` que foi inicializado na linha 125 do Quadro 11, e atribuir à variável `rawData`. Na linha 68 é utilizada a classe `Parsing` para transformar o objeto `rawData` em um objeto do tipo `Parsed`, atribuído à variável `translated`. Em seguida na instrução `switch` é tratado o parâmetro `col` de forma a apresentar o valor apropriado na interface de usuário. Nas linhas 71 e 73 obtém-se respectivamente o PID e o valor em hexadecimal e nas linhas 75 e 78 obtém-se a informação humanamente legível deste PID. Os valores renderizados podem ser observados na Figura 27, apresentada no início desta seção.

A classe `Parsing` utiliza as configurações do arquivo `obd-jrp.properties` para selecionar a implementação apropriada de `Parser` para tratar determinado PID. As propriedades que identificam as implementações de `Parser` seguem o seguinte formato: `parser_<número do PID>=<nome da classe implementadora>`. Exemplos da propriedade `parser` podem ser observados no Quadro 13.

¹⁵ Documentação disponível em <https://docs.oracle.com/javase/8/docs/api/javax/swing/JTable.html>.

Quadro 13 - Configuração dos Parsers

```

46 # nomes das classes Parser dos valores dos PIDs
47 parser_04=br.com.staroski.obdjrj.parsers.CalculatedEngineLoad
48 parser_05=br.com.staroski.obdjrj.parsers.EngineCoolantTemperature
49 parser_06=br.com.staroski.obdjrj.parsers.ShortTermFuelTrim_Bank1
50 parser_07=br.com.staroski.obdjrj.parsers.LongTermFuelTrim_Bank1
51 parser_08=br.com.staroski.obdjrj.parsers.ShortTermFuelTrim_Bank2
52 parser_09=br.com.staroski.obdjrj.parsers.LongTermFuelTrim_Bank2
53 parser_0C=br.com.staroski.obdjrj.parsers.EngineRPM
54 parser_0D=br.com.staroski.obdjrj.parsers.VehicleSpeed
55 parser_0F=br.com.staroski.obdjrj.parsers.IntakeAirTemperature
56 parser_10=br.com.staroski.obdjrj.parsers.MAFAirFlowRate
57 parser_11=br.com.staroski.obdjrj.parsers.ThrottlePosition
58 parser_14=br.com.staroski.obdjrj.parsers.OxygenSensor1
59 parser_15=br.com.staroski.obdjrj.parsers.OxygenSensor2
60 parser_16=br.com.staroski.obdjrj.parsers.OxygenSensor3
61 parser_17=br.com.staroski.obdjrj.parsers.OxygenSensor4
62 parser_18=br.com.staroski.obdjrj.parsers.OxygenSensor5
63 parser_19=br.com.staroski.obdjrj.parsers.OxygenSensor6
64 parser_1A=br.com.staroski.obdjrj.parsers.OxygenSensor7
65 parser_1B=br.com.staroski.obdjrj.parsers.OxygenSensor8

```

Fonte: Elaborado pelo autor.

3.3.1.4 Comunicação com a interface ELM327 Bluetooth

Para efetivamente trocar mensagens com a interface ELM327, a classe `Scanner` utiliza uma instância da classe `ELM327`. Ao instanciar um `Scanner`, o primeiro passo executado é criar uma instância de `ELM327`, em seguida é requisitado ao `ELM327` a lista de PIDs suportados pelo veículo. Esta lista será tomada como base para definir os ciclos de criação de objetos de leitura, representados pela classe `Scan`. No Quadro 14 é apresentado o construtor da classe `Scanner`.

Quadro 14 - Construtor da classe Scanner

```

75 public Scanner(IO connection) throws IOException, ELM327Error {
76     elm327 = startupELM327(connection);
77     supportedPIDs = loadSupportedPIDs();
78
79     this.eventMulticaster = new EventMulticaster();
80     this.scanLoop = new ScanLoop(this);
81 }

```

Fonte: Elaborado pelo autor.

Na linha 76 é invocado o método `startupELM327`, responsável por inicializar uma instância da classe `ELM327` e em seguida, na linha 77, através do método `loadSupportedPIDs`, obtém-se a lista de PIDs suportados pelo veículo. O Quadro 15 apresenta a implementação do método `startupELM327` e o Quadro 16 apresenta a implementação do método `loadSupportedPIDs`. Na linha 79 é inicializado um objeto da classe `EventMulticaster`. Esta classe encapsula uma lista objetos `ScannerListener` e ela própria também implementa a interface `ScannerListener`, dessa forma para a classe `Scanner` a implementação é feita como se só houvesse um único `ScannerListener` registrado. No Quadro 18 é apresentado o código

fonte da classe `EventMulticaster`. Na linha 80 é inicializado um objeto `ScanLoop`, que dispara uma `Thread` a qual permanece num laço contínuo executando as leituras do `ELM327`. No Quadro 19 observa-se a implementação do método `execute` da classe `ScanLoop`.

Quadro 15 - Inicializando objeto ELM327

```
56 private static ELM327 startupELM327(IO connection) throws IOException, ELM327Error {
57     PrintStream log = createLogStream();
58     ELM327 elm327 = new ELM327(connection, log);
59     elm327.execute("ATZ"); // reset
60     elm327.execute("ATE0"); // desligando echo
61     elm327.execute("ATH0"); // desligando envio dos cabeçalhos
62     elm327.execute("ATS0"); // desligando espaços em branco
63     elm327.execute("ATSP0"); // definindo detecção automática de protocolo
64     return elm327;
65 }
```

Fonte: Elaborado pelo autor.

Na linha 59 é executado o comando¹⁶ `ATZ`, responsável por reinicializar a interface `ELM327`. Os comandos seguintes `ATE0`, `ATH0` e `ATS0`, são utilizados para que as mensagens da interface `ELM327` sejam produzidas sem mensagens de eco, sem cabeçalhos de retorno e sem espaços em branco entre os bytes retornados. Essa estratégia torna a transmissão mais rápida e facilita a interpretação das respostas. O comando enviado na linha 63, `ATSP0`, é responsável por fazer com que a própria interface `ELM327` selecione automaticamente o protocolo de comunicação com a ECU do veículo. Após executar estas configurações o objeto instanciado é retornado.

¹⁶ Documentação disponível em: https://www.sparkfun.com/datasheets/Widgets/ELM327_AT_Commands.pdf.

Quadro 16 - Obtendo PIDs suportados pelo veículo

```

105 private List<String> loadSupportedPIDs() throws IOException, ELM327Error {
106     final List<String> reservedPIDs = Arrays.asList(new String[] { //
107         "00", // return range 01-20
108         "20", // return range 21-40
109         "40", // return range 41-60
110         "60", // return range 61-80
111         "80", // return range 81-A0
112         "A0", // return range A1-C0
113         "C0", // return range C1-E0
114         "E0", // return range E1-FF
115     });
116     List<String> supportedPIDs = new ArrayList<>();
117     List<String> validPIDs = Arrays.asList(reservedPIDs.get(0));
118     for (String pid : reservedPIDs) {
119         if (!validPIDs.contains(pid)) {
120             break;
121         }
122         String bitmask = readPIDsBitmask(pid);
123         validPIDs = processBitmask(pid, bitmask);
124         supportedPIDs.addAll(validPIDs);
125     }
126     supportedPIDs.removeAll(reservedPIDs);
127     if (supportedPIDs.isEmpty()) {
128         throw new UnsupportedOperationException("Vehicle doesn't provide any OBD2 PID!");
129     }
130     return supportedPIDs;
131 }

```

Fonte: Elaborado pelo autor.

Com o objeto ELM327 devidamente inicializado, o próximo passo é obter a lista de PIDs suportados pelo veículo. Esse processo é realizado utilizando 8 PIDs reservados que retornam 4 bytes correspondentes à uma máscara de 32 bits onde o índice dos bits ligados, começando do mais significativo, para o menos significativo, correspondem à um PID suportado SAE International (2006, p. 118, tradução nossa). Os 8 PIDs reservados são:

- a) 00, obtém os PIDs válidos entre 01 e 20;
- b) 20, obtém os PIDs válidos entre 21 e 40;
- c) 40, obtém os PIDs válidos entre 41 e 60;
- d) 60, obtém os PIDs válidos entre 61 e 80;
- e) 80, obtém os PIDs válidos entre 81 e A0;
- f) A0, obtém os PIDs válidos entre A1 e C0;
- g) C0, obtém os PIDs válidos entre C1 e E0;
- h) E0, obtém os PIDs válidos entre E1 e FF.

Para exemplificar o processo de reconhecimento de PIDs suportados, será utilizado o seguinte cenário hipotético:

- a) executa-se o PID 00;
- b) obtém como resposta os bytes BEBACAFE;
- c) convertendo binário, obtém-se 10111110101110101100101011111110;
- d) nesta máscara, o bit mais significativo tem peso 1 e o menos significativo, 32;

- e) itera-se sobre os bits da esquerda para a direita, armazenando o peso de cada bit 1;
- f) obtém-se os seguintes valores decimais: 1, 3, 4, 5, 6, 7, 9, 11, 12, 13, 15, 17, 18, 21, 23, 25, 26, 27, 28, 29, 30 e 31;
- g) soma-se cada valor com o PID reservado, em decimal, que neste exemplo é 0;
- h) converte-se cada valor para hexadecimal obtendo a seguinte lista de PIDs suportados: 01, 03, 04, 05, 06, 07, 09, 0B, 0C, 0D, 0F, 11, 12, 15, 17, 19, 1A, 1B, 1C, 1D, 1E e 1F.

O algoritmo ilustrado no exemplo acima, é implementado pelo método `processBitmask`, apresentado no Quadro 17.

Quadro 17 - Processando máscara de bits

```

133 private List<String> processBitmask(String pid, String bytes) throws IOException {
134     List<String> pids = new ArrayList<>();
135     try {
136         int offset = Integer.parseInt(pid, 16) + 1;
137         char[] bitmask = Conversions.hexaToBinary(bytes, 32).toCharArray();
138         for (int i = 0, value = offset; i < bitmask.length; i++, value++) {
139             if (bitmask[i] == '1') {
140                 pids.add(Conversions.decimalToHexa(value, 8));
141             }
142         }
143         return pids;
144     } catch (NumberFormatException e) {
145         return new ArrayList<>();
146     }
147 }

```

Fonte: Elaborado pelo autor.

Quadro 18 - Classe EventMulticaster

```

9  final class EventMulticaster implements ScannerListener {
10
11      private final List<ScannerListener> listeners;
12
13      EventMulticaster() {
14          listeners = new ArrayList<>();
15      }
16
17      @Override
18      public void onError(ELM327Error error) {
19          for (ScannerListener listener : listeners) {
20              listener.onError(error);
21          }
22      }
23
24      @Override
25      public void onScanned(Scan scannedData) {
26          for (ScannerListener listener : listeners) {
27              listener.onScanned(scannedData);
28          }
29      }
30
31      void addListener(ScannerListener listener) {
32          listeners.add(listener);
33      }
34
35      void removeListener(ScannerListener listener) {
36          listeners.remove(listener);
37      }
38  }

```

Fonte: Elaborado pelo autor.

A classe `EventMultiCaster` implementa a interface `ScannerListener` e permite a agregação de outras instâncias de `ScannerListener`, dessa forma, quando o `EventMultiCaster` trata os eventos `onError` e `onScanned` ele delega o evento para as outras instâncias agregadas de `ScannerListener`. A classe `EventMulticaster` é uma forma de implementação do padrão de projeto Composite¹⁷.

¹⁷ Conforme Rocha (2005), Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme.

Quadro 19 - Método execute da classe ScanLoop

```

57 private void execute() throws IOException {
58     while (scanning) {
59         try {
60             System.out.printf("scanning data...%n");
61             Timer timer = new Timer();
62             Scan scan = scanner.scan();
63             int pids = scan.getData().size();
64             long time = timer.elapsed();
65             System.out.printf("%d PIDs read%n", pids);
66             System.out.printf("data scanned in %dms%n", time);
67             scanner.notifyScanned(scan);
68         } catch (ELM327Error error) {
69             scanning = false;
70             Print.message(error);
71             scanner.notifyError(error);
72         }
73     }
74 }

```

Fonte: Elaborado pelo autor.

O método `execute` da classe `ScanLoop` executa continuamente o laço de leitura dos dados da interface `ELM327` e trata de notificar os eventos `onScanned` e `onError` para os objetos `ScannerListener` registrados na classe `Scanner`. Outra característica interessante da classe `ScanLoop`, é que em seu construtor ela registra um `ScannerListener` chamado `ScanUploader`, como pode ser observado no Quadro 20.

Quadro 20 - Construtor da classe ScanLoop

```

17 public ScanLoop(Scanner scanner) {
18     this.scanner = scanner;
19     scanner.addListener(new ScanUploader());
20 }

```

Fonte: Elaborado pelo autor.

A classe `ScanUploader` é responsável por tentar enviar para o servidor, em tempo real, as leituras obtidas. O fato de implementar a interface `ScannerListener` e ser registrada ao `Scanner`, faz com que o `ScanUploader` seja notificado cada vez que um ciclo de leitura é concluído. No Quadro 21 é apresentada parte da implementação da classe `ScanUploader`. Em seu construtor, na linha 16, é inicializado um objeto do tipo `PackagePersister`, responsável por persistir em disco as leituras que não foram possíveis de serem enviadas ao servidor através do método `upload`, que é apresentado no Quadro 22.

Quadro 21 - Tratamento de eventos na classe ScanUploader

```

15 ScanUploader() {
16     this.persister = new PackagePersister();
17 }
18
19 @Override
20 public void onError(ELM327Error error) {
21     persister.persist();
22 }
23
24 @Override
25 public void onScanned(final Scan scan) {
26     Thread thread = new Thread(new Runnable() {
27
28         @Override
29         public void run() {
30             try {
31                 tryUpload(scan);
32             } catch (Exception error) {
33                 Print.message(error);
34             }
35         }
36     }, "ScanUploader_Thread");
37     thread.start();
38 }
39
40 private void tryUpload(Scan scan) {
41     if (upload(scan)) {
42         persister.persist();
43     } else {
44         persister.add(scan);
45     }
46 }

```

Fonte: Elaborado pelo autor.

Quando a classe `ScanUploader` é notificada de um erro, ela trata de invocar o método `persist` da classe `PackagePersister`, de forma que, caso haja algum dado pendente de envio ao servidor, o mesmo seja persistido em disco para poder ser enviado posteriormente ao servidor pelo processo responsável em monitorar os dados pendentes de envio. No caso de uma notificação de leitura de dados, a classe `ScanUploader` dispara uma `Thread` que executa o método `tryUpload`, que tenta enviar os dados lidos ao servidor através do método `upload`. Caso o método `upload` não consiga enviar os dados ao servidor, eles são adicionados ao `PackagePersister`. E caso consiga enviar, é invocado o método `persist` do `PackagePersister`, pois caso hajam dados que previamente não foram enviados, eles são adicionados ao `PackagePersister` e, a partir do momento que o `upload` voltou a conseguir enviar os dados ao servidor, os dados pendentes podem ser persistidos para que o processo responsável por monitorar os dados pendentes os envie o mais breve possível. No Quadro 22 é apresentada a implementação do método `upload`.

Quadro 22 - Método upload da classe ScanUploader

```

48 private boolean upload(Scan dataScan) {
49     System.out.printf("sending scan to server%n");
50     try {
51         String url = Config.get().webServer() + "/exec?cmd=SendData";
52         String vehicle = Config.get().vehicle();
53         byte[] bytes = dataScan.writeTo(new ByteArrayOutputStream()).toByteArray();
54         String scan = Conversions.bytesToHexas(bytes);
55         String[][] params = new String[][] { //
56             new String[] { "vehicle", vehicle }, //
57             new String[] { "scan", scan }, //
58         };
59         boolean accepted = Http.sendGetRequest(url, params);
60         System.out.printf("scan %s by server%n", accepted ? "accepted" : "rejected");
61         return accepted;
62     } catch (Exception error) {
63         Print.message(error);
64         return false;
65     }
66 }

```

Fonte: Elaborado pelo autor.

Na linha 51 é preparado o endereço do recurso a ser acessado no servidor. Entre as linhas 52 e 58 são preparados os parâmetros da requisição. Na linha 59 a leitura atual, representada pelo parâmetro `dataScan`, é enviada ao servidor através de uma requisição HTTP GET. Embora requisições HTTP GET sejam voltadas para obter recursos do servidor, o método `upload` as utiliza pois objetos do tipo `Scan` possuem pouco conteúdo, podendo ser passados por parâmetro no próprio endereço do recurso sendo acessado.

Observa-se ainda, na linha 52, um parâmetro essencial, a identificação do veículo, configurada no arquivo `obd-jrp.properties` através da propriedade `vehicle`. A ideia inicial era obter o Vehicle Identification Number (VIN), a partir da interface ELM327, utilizando o modo de operação 9 e o PID 02, entretanto, nos dois veículos testados este modo de operação não foi suportado. Dessa forma, optou-se em identificar o veículo através de uma propriedade no arquivo.

A classe `PackagePersister`, utilizada pelo `ScanUploader` procura manter as leituras pendentes em memória até que seja explicitamente invocado o método `persist`, ou até que seja atingida uma quota de leituras para o pacote de dados a ser enviado. Essa quota é definida no arquivo `obd-jrp.properties` através da propriedade `data_max_scans`. No Quadro 23 é apresentada a implementação do método `add` da classe `PackagePersister`.

Quadro 23 - Método add da classe PackagePersister

```

30 public void add(Scan scan) {
31     final int limit = Config.get().dataMaxScans();
32     List<Scan> scannedData = dataPackage.getScans();
33     System.out.printf("adding data %d of %d to data package...%n",
34                       scannedData.size() + 1,
35                       limit);
36     scannedData.add(scan);
37     if (scannedData.size() == limit) {
38         System.out.printf("data package filled!%n");
39         persist();
40     }
41 }

```

Fonte: Elaborado pelo autor.

3.3.1.5 Monitoramento dos dados pendentes de envio

O terceiro programa desenvolvido para o firmware foi o `ObdJrpUploadData`, responsável por monitorar o diretório onde são armazenadas as leituras pendentes de envio. O caminho deste diretório é configurado através da propriedade `data_dir`, no arquivo `obd-jrp.properties`. A classe `ObdJrpUploadData` simplesmente instancia um objeto do tipo `DataMonitor` e invoca o seu método `start`. O `DataMonitor` dispara uma `Thread` que executa continuamente um laço que verifica se há arquivos em disco pendentes de envio, se houver tenta enviá-los ao servidor, caso contrário aguarda 5 minutos no método `end` e tenta novamente. No Quadro 24 é apresentado o método `execute` da classe `DataMonitor`.

Quadro 24 - Método execute da classe DataMonitor

```

91 private void execute() {
92     do {
93         begin();
94         Queue<File> files = getFiles();
95         if (files.isEmpty()) {
96             System.out.println("no data to upload!");
97             end();
98             continue;
99         }
100         for (File file : files) {
101             try {
102                 if (upload(file)) {
103                     file.delete();
104                 }
105             } catch (Throwable error) {
106                 Print.message(error);
107             }
108             if (!scanning) {
109                 return;
110             }
111         }
112         end();
113     } while (scanning);
114 }

```

Fonte: Elaborado pelo autor.

Na linha 94 obtém-se uma lista com os arquivos pendentes. Se a lista estiver vazia, na linha 95, aguarda 5 minutos e volta ao início do laço. Se houve arquivos pendentes, tenta enviá-los ao servidor através do método `upload`, na linha 102. Caso o arquivo seja enviado com sucesso ao servidor, ele é apagado do disco local, na linha 103. No Quadro 25 observa-se a implementação do método `upload` da classe `DataMonitor`.

Quadro 25 - Método `upload` da classe `DataMonitor`

```
138 private boolean upload(File file) {  
139     System.out.printf("uploading file \"%s\"%n", file.getAbsolutePath());  
140     try {  
141         String url = Config.get().webServer() + "/exec?UploadData";  
142         boolean accepted = Http.sendPostRequest(url, file);  
143         System.out.printf("file %s by server%n", accepted ? "accepted" : "rejected");  
144         return accepted;  
145     } catch (Exception error) {  
146         Print.message(error);  
147         return false;  
148     }  
149 }
```

Fonte: Elaborado pelo autor.

Na linha 142 observa-se que o método utilizado para o envio dos arquivos é o HTTP POST, tendo em vista que os arquivos com dados pendentes são maiores do que uma simples leitura, então como não é possível enviá-los através dos parâmetros de uma requisição GET, é necessário preparar um cabeçalho diferenciado para enviar um arquivo grande. Essa operacionalidade é abstraída através do método `Http.sendPostRequest`, a qual cria um objeto do tipo `PostRequest` cujo código é apresentado no Quadro 26.

Quadro 26 - Classe PostRequest

```

24- PostRequest(String requestURL) throws IOException {
25     boundary = "===" + System.currentTimeMillis() + "===";
26     URL url = new URL(requestURL);
27     httpConn = (URLConnection) url.openConnection();
28     httpConn.setUseCaches(false);
29     httpConn.setDoOutput(true);
30     httpConn.setDoInput(true);
31     httpConn.setRequestProperty("Content-Type", "multipart/form-data; boundary="
32         + boundary);
33     httpConn.setRequestProperty("User-Agent", "CodeJava Agent");
34     httpConn.setRequestProperty("Test", "Bonjour");
35     outputStream = httpConn.getOutputStream();
36     writer = new PrintWriter(new OutputStreamWriter(outputStream, charset), true);
37 }
38
39- public void addFilePart(String fieldName, File uploadFile) throws IOException {
40     String fileName = uploadFile.getName();
41     writer.append("--" + boundary).append(LINE_FEED);
42     writer.append("Content-Disposition: form-data; name=\""
43         + fieldName + "\"; filename=\""
44         + fileName + "\"").append(LINE_FEED);
45     writer.append("Content-Type: " + URLConnection.guessContentTypeFromName(fileName))
46         .append(LINE_FEED);
47     writer.append("Content-Transfer-Encoding: binary").append(LINE_FEED);
48     writer.append(LINE_FEED);
49     writer.flush();
50     FileInputStream inputStream = new FileInputStream(uploadFile);
51     byte[] buffer = new byte[4096];
52     int bytesRead = -1;
53     while ((bytesRead = inputStream.read(buffer)) != -1) {
54         outputStream.write(buffer, 0, bytesRead);
55     }
56     outputStream.flush();
57     inputStream.close();
58     writer.append(LINE_FEED);
59     writer.flush();
60 }

```

Fonte: Elaborado pelo autor.

3.3.1.6 Processamento das requisições no servidor

O ponto de entrada na implementação do servidor é a classe `ObdJrpServlet`, responsável por interceptar requisições HTTP GET e HTTP POST. A classe implementa o padrão `Command` e dessa forma delega responsabilidades para os objetos apropriados de acordo com os parâmetros recebidos. O corpo da classe `ObdJrpServlet` é apresentado no Quadro 27.

Quadro 27 - Classe ObdJrpServlet

```

25     private static final Map<String, Command> COMMANDS = new HashMap<>();
26
27     static {
28         COMMANDS.put("ListVehicles", new ListVehicles());
29         COMMANDS.put("ReadData", new ReadData());
30         COMMANDS.put("SendData", new SendData());
31         COMMANDS.put("UploadData", new UploadData());
32         COMMANDS.put("ViewChart", new ViewChart());
33     }
34
35     private Command getCommand(HttpServletRequest request) {
36         Command command = COMMANDS.get(request.getParameter("cmd"));
37         return command == null ? Command.NULL : command;
38     }
39
40     private void process(HttpServletRequest request, HttpServletResponse response) {
41         Command command = getCommand(request);
42         String result = command.execute(request, response);
43         if (result != null) {
44             response.setHeader("Cache-Control", "no-cache");
45             request.getRequestDispatcher(result).forward(request, response);
46         }
47     }
48
49     protected void doGet(HttpServletRequest request, HttpServletResponse response) {
50         process(request, response);
51     }
52
53     @Override
54     protected void doPost(HttpServletRequest request, HttpServletResponse response) {
55         process(request, response);
56     }

```

Fonte: Elaborado pelo autor.

Entre as linhas 27 e 33 é inicializado um mapa com os recursos disponibilizados pela classe ObdJrpServlet, os métodos doGet e doPost delegam sua execução para o método process, que obtém uma instância de Command a partir dos parâmetros da requisição recebida. O Command é executado e, caso tenha retornado uma página de resposta, a requisição é redirecionada para esta página.

As classes SendData e UploadData, são os recursos acessados pelo firmware para respectivamente enviar leituras em tempo real e para realizar o upload de arquivos com leituras pendentes. As classes ListVehicles, ReadData e ViewChart, são os recursos acessados pelo usuário ao consultar informações pelo navegador web. No Quadro 28 é apresentada a implementação da classe SendData e no Quadro 29 é apresentada a implementação da classe UploadData.

Quadro 28 - Command SendData

```

20 @Override
21 public String execute(HttpServletRequest request, HttpServletResponse response) throws
22     PrintWriter out = response.getWriter();
23     try {
24         String vehicleId = request.getParameter("vehicle");
25         byte[] scanBytes = Conversions.hexasToBytes(request.getParameter("scan"));
26         saveScan(vehicleId, Scan.readFrom(new ByteArrayInputStream(scanBytes)));
27         ObdJrpWeb.organizeScanDir(vehicleId);
28         out.println("OK");
29     } catch (Exception error) {
30         System.out.printf("%s: %s\n", //
31             error.getClass().getSimpleName(), //
32             error.getMessage());
33         out.println("ERROR");
34     }
35     return null;
36 }
37
38 private boolean saveScan(String vehicleId, Scan scan) {
39     try {
40         File scanFile = getFile(vehicleId, scan, ".scan");
41         System.out.printf("Saving \"%s\"...\n", scanFile.getAbsolutePath());
42         FileOutputStream scanOutput = new FileOutputStream(scanFile);
43         scan.writeTo(scanOutput);
44         scanOutput.close();
45         scanFile.setLastModified(scan.getTime());
46         System.out.println("OK!");
47         return true;
48     } catch (IOException e) {
49         System.out.println("ERRO!");
50         e.printStackTrace();
51         return false;
52     }
53 }

```

Fonte: Elaborado pelo autor.

Quadro 29 - Command UploadData

```

23 @Override
24 public String execute(HttpServletRequest request, HttpServletResponse response) throws
25     PrintWriter out = response.getWriter();
26     Part uploadedFile = request.getPart("fileUpload");
27     if (uploadedFile != null && savePart(uploadedFile)) {
28         out.println("OK");
29     } else {
30         out.println("ERROR");
31     }
32     response.setStatus(HttpURLConnection.HTTP_OK);
33     return null;
34 }
35
36 private boolean savePart(Part part) {
37     try {
38         InputStream input = part.getInputStream();
39         Package dataPackage = Package.readFrom(input);
40         return saveDataPackage(dataPackage);
41     } catch (Exception e) {
42         e.printStackTrace();
43         return false;
44     }
45 }

```

Fonte: Elaborado pelo autor.

A classe `ReadData` permite a visualização dos dados lidos em tempo real em uma página que renderiza uma tabela semelhante à tela implementada no programa `ObdJrpScanData`, que é executado no firmware. `ReadData` também é responsável por apresentar uma página com o

histórico de até 1000 leituras de determinado PID. A implementação da classe ReadData é apresentada no Quadro 30.

Quadro 30 - Command ReadData

```

14 @Override
15 public String execute(HttpServletRequest request, HttpServletResponse response) throws :
16     String vehicleId = request.getParameter("vehicle");
17     if (vehicleId == null || vehicleId.isEmpty()) {
18         return null;
19     }
20
21     String history = request.getParameter("history");
22     if ("yes".equalsIgnoreCase(history)) {
23         String pid = request.getParameter("pid");
24         List<Scan> scans = ObdJrpWeb.getAllScans(vehicleId);
25         request.setAttribute("vehicle", vehicleId);
26         request.setAttribute("pid", pid);
27         request.setAttribute("history_table_model", new HistoryTableModel(pid, scans));
28         return "vehicle-history.jsp";
29     }
30
31     Scan lastScan = ObdJrpWeb.getLastScan(vehicleId);
32     request.setAttribute("vehicle", vehicleId);
33     request.setAttribute("scan_table_model", new ScanTableModel(lastScan));
34     return "vehicle-detail.jsp";
35 }

```

Fonte: Elaborado pelo autor.

Observa-se na linha 27 a disponibilização de um objeto do tipo `HistoryTableModel` e na linha 33 um objeto do tipo `ScanTableModel`. Estas classes foram baseadas na interface `javax.swing.table.TableModel` e são utilizadas respectivamente pelas páginas `vehicle-history.jsp` e `vehicle-detail.jsp`, para renderizar a grade com os campos e valores lidos. No Quadro 31 é apresentada a implementação da página `vehicle-history.jsp` e no Quadro 32 é apresentada a implementação da página `vehicle-detail.jsp`. Observa-se que o uso das classes `HistoryTableModel` e `ScanTableModel` tornou idênticas as implementações das duas páginas.

Quadro 31 - Página vehicle-history.jsp

```

16  final HistoryTableModel model = (HistoryTableModel) request.getAttribute("history_table_model");
17  final int rows = model.getRowCount();
18  final int columns = model.getColumnCount();
19  final int lastColumn = columns - 1; %>
20  <div id="title" align="center">
21      <h1>Vehicle</h1>
22      <h2><%=vehicle%></h2>
23  </div>
24  <div id="title" align="center">
25      <h2>PID <%=pid%> History</h2>
26  </div>
27  <div id="scan-data" align="center">
28      <div class="divTable">
29          <div class="divTableBody">
30              <div class="divTableRow">
31                  <% for (int j = 0; j < columns; j++) { %>
32                      <div class="divTableCell" align="center"><%=model.getColumnName(j)%></div>
33                  <% } %>
34              </div>
35              <% for (int i = 0; i < rows; i++) { %>
36                  <div class="divTableRow">
37                      <% for (int j = 0; j < columns; j++) {
38                          String alignment = j == lastColumn ? "right" : "left";
39                          String value = (String) model.getValueAt(i, j); %>
40                          <div class="divTableCell" align=<%=alignment%>><%=value%></div>
41                      <% } %>
42                  </div>
43              <% } %>
44          </div>
45      </div>
46  </div>

```

Fonte: Elaborado pelo autor.

Observa-se que, entre as linhas 31 e 43 do Quadro 31, ocorre a manipulação do objeto `HistoryTableModel` de forma a construir as tags HTML para a renderização da grade com os campos e valores. Processo semelhante ocorre entre as linhas 29 e 46 do Quadro 32.

Quadro 32 - Página vehicle-detail.jsp

```

17 final ScanTableModel model = (ScanTableModel) request.getAttribute("scan_table_model");
18 final int rows = model.getRowCount();
19 final int columns = model.getColumnCount();
20 final int lastColumn = columns - 1; %>
21 <div id="title" align="center">
22     <h1>Vehicle</h1>
23     <h2><%=vehicle%></h2>
24 </div>
25 <div id="scan-data" align="center">
26     <div class="divTable">
27         <div class="divTableBody">
28             <div class="divTableRow">
29                 <% for (int j = 0; j < columns; j++) { %>
30                     <div class="divTableCell" align="center"><%=model.getColumnName(j)%></div>
31                 <% } %>
32             </div>
33             <% for (int i = 0; i < rows; i++) {
34                 String pid = (String) model.getValueAt(i, 0); %>
35                 <div class="divTableRow">
36                     <% for (int j = 0; j < columns; j++) {
37                         String alignment = j == lastColumn ? "right" : "left";
38                         String value = (String) model.getValueAt(i, j);
39                         if (j == 2 && model.hasParser(i)) {
40                             value = "<a href=\"exec?cmd=ViewChart&vehicle="
41                                 + vehicle + "&pid=" + pid + "\">" + value + "</a>";
42                         } %>
43                     <div class="divTableCell" align=<%=alignment%>><%=value%></div>
44                     <% } %>
45                 </div>
46             <% } %>
47         </div>
48     </div>
49 </div>

```

Fonte: Elaborado pelo autor.

A classe `ViewChart` é responsável por fornecer para a página Java Server Pages (JSP) um objeto do tipo `ChartBuilder`, que possui métodos para criar as tags HTML e JavaScript necessárias para renderização de gráficos na página. No Quadro 33 é apresentada a implementação da classe `ViewChart` e no Quadro 34 é apresentado o código fonte da página `view-chart.jsp`.

Quadro 33 - Command ViewChart

```

14 public String execute(HttpServletRequest request, HttpServletResponse response) throws
15     String vehicleId = request.getParameter("vehicle");
16     String pid = request.getParameter("pid");
17     String chart = request.getParameter("chart");
18     if (vehicleId == null || vehicleId.isEmpty()) {
19         return null;
20     }
21     if (pid == null || pid.isEmpty()) {
22         return null;
23     }
24
25     Scan lastScan = ObdJrpWeb.getLastScan(vehicleId);
26     ChartBuilder chartBuilder = new ChartBuilder(vehicleId, pid);
27
28     if (chart != null && !chart.isEmpty()) {
29         String[] charts = chart.split(",");
30         StringBuilder builder = new StringBuilder();
31         builder.append("{ \"items\":[");
32         for (int i = 0; i < charts.length; i++) {
33             if (i > 0) {
34                 builder.append(",\n");
35             }
36             String data = chartBuilder.createChartData(Integer.parseInt(charts[i]));
37             builder.append(data);
38         }
39         builder.append("]");
40         response.getWriter().write(builder.toString());
41         return null;
42     }
43
44     request.setAttribute("vehicle", vehicleId);
45     request.setAttribute("pid", pid);
46     request.setAttribute("scan_time", lastScan.getTime());
47     request.setAttribute("chart_builder", chartBuilder);
48     return "vehicle-chart.jsp";
49 }

```

Fonte: Elaborado pelo autor.

Na linha 47 é fornecido o objeto `ChartBuilder` para a página JSP renderizar os gráficos.

Quadro 34 - Página view-chart.jsp

```

1  <%@page import="br.com.staroski.obdjrp.web.ChartBuilder"%>
2  <%@page import="java.util.Date"%>
3  <%@page import="java.text.SimpleDateFormat"%>
4  <%@page contentType="text/html"%>
5  <%@page pageEncoding="UTF-8"%>
6  <%
7      final String vehicle = (String) request.getAttribute("vehicle");
8      final String pid = (String) request.getAttribute("pid");
9      final ChartBuilder chart_builder = (ChartBuilder) request.getAttribute("chart_builder");
10 %>
11 <!DOCTYPE HTML PUBLIC>
12 <html>
13 <head>
14 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
15 <title>OBD-JRP</title>
16 <%=chart_builder.createTagScript()%>
17 </head>
18 <body>
19     <div id="title" align="center">
20         <h1>Vehicle</h1>
21         <h2><%=vehicle%></h2>
22     </div>
23     <form action="exec">
24         <input type="hidden" name="cmd" value="ReadData" />
25         <input type="hidden" name="vehicle" value="<%=vehicle%>" />
26         <input type="hidden" name="history" value="yes" />
27         <input type="hidden" name="pid" value="<%=pid%>" />
28         <div align="center">
29             <input type="submit" value="View History" />
30         </div>
31     </form>
32     <div align="center">
33         <%=chart_builder.createTagDiv()%>
34     </div>
35 </body>
36 </html>

```

Fonte: Elaborado pelo autor.

Na linha 16 observa-se o fragmento de código onde é utilizado o objeto `ChartBuilder` para produzir as tags JavaScript e na linha 33 o objeto `ChartBuilder` é utilizado para produzir as tags HTML onde serão renderizados os gráficos.

`ListVehicles` é um `Command` que foi utilizado apenas para testes, sendo que o usuário das páginas web não tem acesso direto a ele. Ele gera uma página JSP com a lista de veículos que possuem dados no servidor, essa lista é montada a partir dos diretórios de leituras persistidas. Observa-se, no Quadro 35 a implementação da classe `ListVehicles`.

Quadro 35 - Command ListVehicles

```

12 class ListVehicles implements Command {
13
14     @Override
15     public String execute(HttpServletRequest request, HttpServletResponse response)
16     {
17         List<String> vehicles = new ArrayList<>();
18         File dir = ObdJrpWeb.getDataDir();
19         for (File f : dir.listFiles()) {
20             vehicles.add(f.getName());
21         }
22         request.setAttribute("vehicles", vehicles);
23         return "vehicle-list.jsp";
24     }
25 }

```

Fonte: Elaborado pelo autor.

3.3.2 Operacionalidade da implementação

Para a execução o firmware conecta-se a interface ELM327 Bluetooth na porta OBD2 do veículo e o Raspberry Pi no conector do acendedor de cigarros utilizando um adaptador USB veicular. Para conectar o Raspberry Pi à internet, sem depender de WiFi, foi utilizado um mini modem USB com chip de dados 4G. As figuras a seguir, expõe as diferentes localizações do conector OBD2 e do conector do acendedor de cigarros.

Na Figura 29 observa-se as características do Gol, onde o conector OBD2 fica situado ao lado esquerdo do motorista, próximo ao painel de fusíveis, já o conector do acendedor de cigarros fica situado no porta-objetos, em frente ao passageiro.

Figura 29 - Instalação no Volkswagen Gol 2010



Fonte: Elaborado pelo autor.

Na Figura 30 observa-se que o conector OBD2 fica situado no próprio painel de fusíveis, abaixo do volante e o conector do acendedor de cigarros fica situado no porta-objetos, próximo ao câmbio.

Figura 30 - Instalação no Volkswagen SpaceFox 2009



Fonte: Elaborado pelo autor.

A Figura 31 apresenta o aspecto do adaptador USB veicular utilizado. Optou-se por este modelo, da marca Energizer, pois o mesmo dispõe de duas conexões USB e cada uma delas

pode alimentar cargas que necessitem de até 2,4 ampères, sendo suficiente para alimentar o Raspberry Pi.

Figura 31 - Aspecto do adaptador USB veicular



Fonte: Elaborado pelo autor.

A Figura 32 apresenta o aspecto do mini modem USB utilizado, trata-se de um modem da marca Huawei (Embora perceba-se o logotipo da operadora TIM, o modem é desbloqueado e o chip de dados 4G utilizado foi adquirido com a operadora Oi).

Figura 32 - Aspecto do mini modem USB



Fonte: Elaborado pelo autor.

Com os dispositivos devidamente conectados ao veículo, o usuário interessado em monitorá-lo, acessa a página web¹⁸ e visualiza a página inicial, apresentada na Figura 33. Nesta página ele informa o chassi¹⁹ do veículo e seleciona o botão “View”.

¹⁸ Para disponibilizar o acesso ao servidor, utilizou-se o serviço No-IP, disponível em: <https://www.noip.com>.

¹⁹ Para os testes realizados, não foi utilizado o número real do chassi do veículo, ao invés disto, utilizou-se dígitos repetidos como 0000000000000000, 1111111111111111 e 2222222222222222.

Figura 33 - Página inicial



Fonte: Elaborado pelo autor.

Após informar o chassi do veículo, o usuário é redirecionado para a página que mostra as leituras em tempo real, como pode ser observado na Figura 34. Nesta página, o usuário pode selecionar os links²⁰ dos PIDs, de forma a obter uma visualização gráfica atualizada em tempo real, como pode ser observado na Figura 35.

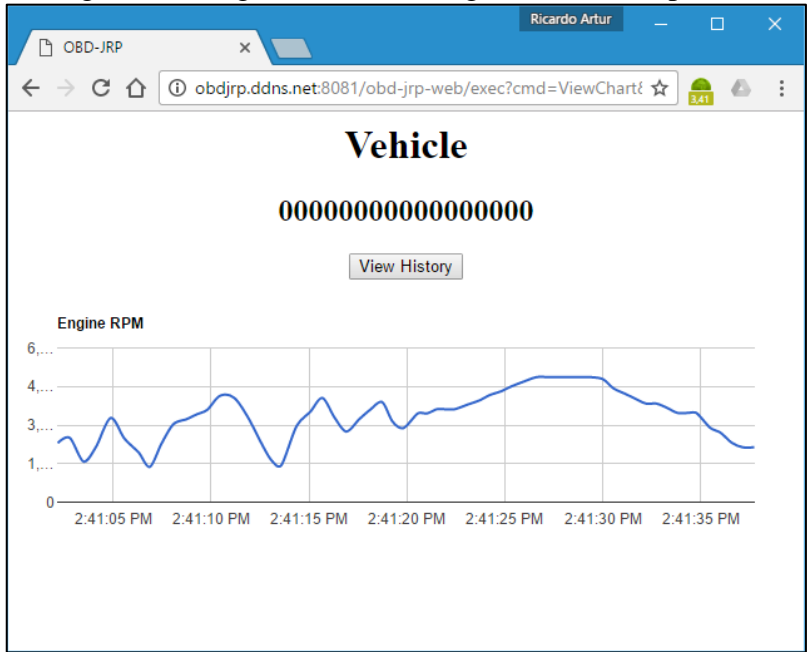
Figura 34 - Página com leituras em tempo real

PID	Bytes	Description	Value
01	8517FF00		
04	EE	Calculated engine load	93,33
05	A5	Engine coolant temperature	125
06	40	Short term fuel trim Bank 1	-50,00
07	E0	Long term fuel trim Bank 1	75,00
08	B0	Short term fuel trim Bank 2	37,50
09	FF22	Long term fuel trim Bank 2	99,22
0C	04B0	Engine RPM	300
0D	1E	Vehicle Speed	30
0F	5F	Intake Air Temperature	55
10	6D7E	MAF Air Flow Rate	280

Fonte: Elaborado pelo autor.

²⁰ Os links são renderizados somente para os PIDs dos quais foi implementado um Parser para obtenção de informação humanamente legível.

Figura 35 - Página com leituras gráficas em tempo real



Fonte: Elaborado pelo autor.

Nesta página é possível visualizar o histórico de leituras do PID em questão, para isso é necessário que o usuário selecione o botão “View History”. A página com o histórico é apresentada na Figura 36. Diferentemente da página de gráficos e da página de leitura em tempo real, a página de histórico não é atualizada automaticamente, pois ela apresenta somente as últimas 1000 leituras realizadas até o momento em que foi selecionado o botão “View History”.

Figura 36 - Página com histórico de leituras

Vehicle

00000000000000000000

PID 0C History

Time	Bytes	Value
2016-11-25-14-43-54-046	2FE4	3065
2016-11-25-14-43-53-280	177C	1503
2016-11-25-14-43-52-504	365C	3479
2016-11-25-14-43-51-757	2D20	2888
2016-11-25-14-43-51-153	1344	1233
2016-11-25-14-43-50-372	2B68	2778
2016-11-25-14-43-49-620	317C	3167
2016-11-25-14-43-48-908	0BBC	751
2016-11-25-14-43-48-257	1D28	1866
2016-11-25-14-43-47-576	3144	3268

Fonte: Elaborado pelo autor.

3.4 RESULTADOS E DISCUSSÕES

O tempo de resposta entre o envio dos dados do firmware e a atualização das páginas no servidor se mostrou aceitável²¹. O firmware foi testado em 3 veículos, um GM Corsa Sedan 2003, Volkswagen Gol 2010 e um Volkswagen SpaceFox 2009. Observou-se que o Corsa Sedan 2003, embora possua o conector, não implementa nenhum protocolo OBD2, não sendo possível o seu monitoramento através do protótipo desenvolvido.

Ao testar a primeira vez no Gol 2010, foi necessário reescrever boa parte dos códigos das classes responsáveis pela leitura da interface ELM327. Isto foi necessário pois as respostas do Gol eram obtidas com espaços em branco, cabeçalhos e mensagens de eco.

Quando a interface ELM327 Bluetooth é ligada ao conector OBD2, imediatamente ela fica disponível para descoberta na rede Bluetooth, entretanto, não é possível se conectar à mesma quando a chave de ignição está desligada, pois a própria interface ELM327 retorna uma mensagem de erro `UNABLE TO CONNECT`, que na implementação desenvolvida, é transformada em uma exceção e lançada adiante.

Quando a chave de ignição é ligada, a interface ELM327 passa a responder de acordo com os comandos enviados, entretanto, ao desligar a chave de ignição, a interface ELM327 continua respondendo com a mensagem `NO DATA`, por isso reescreveu-se o ciclo de leituras de forma que caso um ciclo inteiro obtenha respostas `NO DATA`, o próprio software dispara uma exceção desconectando o link Bluetooth e voltando ao ponto de entrada tentando reestabelecer a conexão com a interface ELM327.

Observou-se que após desligar e ligar a chave de ignição várias vezes, e ao abortar o firmware repentinamente, a interface ELM327 se torna instável, trazendo respostas de comandos anteriores.

Foram comparados os PIDs suportados entre o Gol e o SpaceFox, sendo o comparativo apresentado no Quadro 36.

²¹ Vídeos dos testes disponibilizados em: <https://goo.gl/b3GmGd>.

Quadro 36 - PIDs suportados do Gol e SpaceFox

PID	Gol	SpaceFox
01	X	X
03	X	X
04	X	X
05	X	X
06	X	
07	X	
0B	X	X
0C	X	X
0D	X	X
0E	X	X
0F	X	X
11	X	X
13	X	X
14	X	
15	X	
1C	X	X
21	X	X
51	X	X

Fonte: Elaborado pelo autor.

O Volkswagen SpaceFox 2009 apresentou 4 PIDs a menos que o Volkswagen Gol 2010. Os PIDs 06 e 07 são pertinentes à sensores de combustível e os PIDs 14 e 15 são pertinentes a sensores de oxigênio. Com essa diferença fica evidente que estes dois veículos, embora sejam do mesmo fabricante, possuem características diferentes.

O Quadro 37 apresenta um comparativo entre as características mais relevantes dos trabalhos correlatos apresentados e as características do trabalho desenvolvido.

Quadro 37 - Comparativo entre os trabalhos correlatos e o trabalho desenvolvido

Características mais relevantes	Trabalhos Correlatos		Staroski (2016)
	PyOBD	EnviroCar	OBD-JRP
Funciona em smartphone (Sistema Android)		X	
Funciona em notebook / desktop PC (Sistemas Windows / Linux)	X		X
Funciona em dispositivo Raspberry Pi (Sistema Raspbian)	X		X
Conecta-se à porta OBD2 através de interface ELM327 USB	X		
Conecta-se à porta OBD2 através de interface ELM327 Bluetooth		X	X
Permite visualização dos dados OBD2 em tempo real	X	X	X
Faz upload dos dados coletados para um servidor externo		X	X
Disponibiliza histórico dos dados coletados em página web		X	X
Disponibiliza gráficos dos dados coletados em página web			X

Fonte: Elaborado pelo autor.

Observa-se no Quadro 37 que o trabalho desenvolvido além de disponibilizar visualização dos dados coletados através de gráficos, procura combinar características distintas dos trabalhos correlatos, exceto o suporte à plataforma Android e a conexão por interface ELM327 USB.

4 CONCLUSÕES

O protótipo desenvolvido atendeu aos objetivos propostos. O Raspberry Pi demonstrou ser um equipamento de preço acessível (a partir de R\$120,00), robusto e com capacidade computacional análoga à de um PC, sendo viável para o desenvolvimento de aplicações IOT. A biblioteca BlueCove demonstrou que é viável utilizar a plataforma Java para a comunicação Bluetooth em um computador de arquitetura ARM. A interface ELM327 Bluetooth demonstrou ser um equipamento de custo acessível (cerca de R\$30,00) para realizar a integração entre software e veículo.

Após testar o protótipo em veículos reais, destacaram-se as seguintes vantagens:

- a) dimensões reduzidas;
- b) fácil instalação em qualquer veículo que implemente OBD2;
- c) independência de interação humana após instalado;
- d) permite o monitoramento a partir de qualquer computador ou dispositivo móvel com acesso à internet.

Também foram observadas algumas limitações, das quais destacam-se:

- a) a necessidade de pré-configurar o acesso à internet;
- b) a ausência de um recurso para desligar o dispositivo, existindo o risco de corromper o sistema de arquivos ao desconectar a fonte de alimentação;
- c) a não utilização de um sistema gerenciador de banco de dados torna vulnerável a integridade dos dados gravados em arquivo.

4.1 EXTENSÕES

Para dar continuidade ao trabalho, sugere-se as seguintes extensões:

- a) integrar o protótipo com um banco de dados que comporte alto volume de dados;
- b) integrar o protótipo com um sistema cognitivo que possa diagnosticar antecipadamente possíveis avarias ou manutenções a partir do histórico dos PIDs;
- c) desenvolver hardware para desligar o protótipo quando o veículo é desligado ou quando o usuário desejar;
- d) prover uma interface de usuário no próprio dispositivo, transformando-o em um computador de bordo;
- e) integrar o protótipo com GPS, de forma a também ser possível monitorar sua localização geográfica.

REFERÊNCIAS

- ALDERTON, L. Raspberry Pi - Bluetooth using Bluecove on Raspbian. **Luke Alderton**, p. 1, 3 jan 2015. Disponível em: <<http://lukealderton.com/blog/posts/2015/january/raspberry-pi-bluetooth-using-bluecove-on-raspbian.aspx>>. Acesso em: 13 ago. 2016.
- ANDROID PIT INTERNATIONAL. Android Apps - Lifestyle. **Android Pit International**, [S.l.], p. 1, 2016. Disponível em: <<https://www.androidpit.com/app/org.envirocar.app>>. Acesso em: 20 maio 2016.
- BLUECOVE. **BlueCove**, p. 1, 2008. Disponível em: <<http://www.bluecove.org>>. Acesso em: 20 out. 2016.
- BLUETOOTH. Specification of the Bluetooth System. **Bluetooth**, p. 1907-1967, 2014. Disponível em: <https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=286439>. Acesso em: 05 ago. 2016.
- CONAMA. Resolução CONAMA nº 354, de 13 de dezembro de 2004. Publicada no D.O.U. nº 239, de 14 de dezembro de 2004, Seção 1, p. 62-63, 2004. Disponível em: <http://www.mma.gov.br/port/conama/legislacao/CONAMA_RES_CONS_2004_354.pdf>. Acesso em: 20 maio 2016.
- ELM ELECTRONICS. OBD to RS232 Interpreter. **ELM Electronics - Circuits for the Hobbyist**, [S.l.], p. 1-94, 2016. Disponível em: <<http://www.elmelectronics.com/DSheets/ELM327DS.pdf>>. Acesso em: 20 maio 2016.
- ENVIROCAR. Off we go. **EnviroCar**, [S.l.], p. 1, 2015. Disponível em: <<http://envirocar.org>>. Acesso em: 20 maio 2016.
- GSM ASSOCIATION. Understanding the Internet of Things (IoT). **GSM Association**, [S.l.], p. 1, 2014. Disponível em: <http://www.gsma.com/connectedliving/wp-content/uploads/2014/08/cl_iot_wp_07_14.pdf>. Acesso em: 20 maio 2016.
- HORTA, W. Sonda lambda: controlando a mistura ar-combustível. **Best Cars Web Site**, p. 1, 2000. Disponível em: <<http://bestcars.uol.com.br/ct/lambda.htm>>. Acesso em: 02 ago. 2016.
- MANAVELLA, H. J. Diagnóstico Automotivo Avançado. **HM Autotrônica**, [S.l.], p. 121-127, 2009. Disponível em: <<http://www.hmautotron.eng.br/zip/cap19-hm004web.pdf>>. Acesso em: 20 maio 2016.
- NEW IT LIMITED. RPi 3 (2016) Model B. **New IT Limited**, [S.l.], p. 1, 2016. Disponível em: <https://www.newit.co.uk/shop/all-raspberry-pi/raspberry_pi_3/raspberry_pi3>. Acesso em: 20 maio 2016.
- NG, A. The Evolution of the "Internet of Things": from "Diagnostics and Repair" to "Prescriptive and Proactive". **Horton Works**, [S.l.], p. 1, 2015. Disponível em: <<http://br.hortonworks.com/blog/the-evolution-of-the-internet-of-things-from-diagnostics-and-repair-to-prescriptive-and-proactive>>. Acesso em: 20 maio 2016.
- OUTILS OBD FACILE. Automotive Electronic Diagnostic. **Outils OBD Facile**, [S.l.], p. 1, 2015. Disponível em: <<http://www.outilsobdfacile.com/obd-mode-pid.php>>. Acesso em: 20 maio 2016.
- PYOB. Open Source OBD2 Diagnostics. **OBD Tester**, [S.l.], p. 1, 2015. Disponível em: <<http://www.obdtester.com/pyobd>>. Acesso em: 20 maio 2016.

RASPBERRY PI FOUNDATION. Teach, Learn and make with Raspberry Pi. **Raspberry Pi Foundation**, [S.l.], p. 1, 2016. Disponível em: <<https://www.raspberrypi.org>>. Acesso em: 20 maio 2016.

RIORAND. On Board Diagnostics. **RioRand Advanced Technology**, [S.l.], p. 1, 2015. Disponível em: <<http://www.riorand.com/on-board-diagnostics>>. Acesso em: 20 maio 2016.

ROCHA, H. D. Padrões de Design com aplicações em Java. **Argonavis**, p. 1-223, 2005. Disponível em: <http://www.argonavis.com.br/cursos/java/j930/tutorial/Design_Patterns.pdf>. Acesso em: 03 set. 2016.

SAE INTERNATIONAL. Society of Automotive Engineers. **DocumBase.com**, [S.l.], p. 1-228, 2006. Disponível em: <<http://www.documbase.com/goto/9552188-77674d95ff4a7fd13cf79446a8561c94/SAE-J1979-2006-edition-Ballot.pdf>>. Acesso em: 20 maio 2016.

THE BEST OBD2 SCANNERS. 10 Modes of Operation for OBD2 Scanners. **The Best OBD2 Scanners**, [S.l.], p. 1, 2016. Disponível em: <<http://thebestobdiiscanners.com/10-modes-of-operation-for-obd2-scanners>>. Acesso em: 20 maio 2016.

THOMPSON, T. J.; KLINE, P. J.; KUMAR, C. B. Bluetooth Application Programming with the Java APIs. Burlington: Morgan Kaufmann Publishers, 2008. p. 23-34. ISBN 978-0-12-374342-8. Disponível em: <https://www.academia.edu/attachments/33201269/download_file>. Acesso em: 20 out. 2016.

THOMSEN, A. Saiu o Raspberry Pi 3. **Filipe Flop Componentes Eletrônicos**, [S.l.], p. 1, 2016. Disponível em: <<http://blog.filipeflop.com/embarcados/saiu-o-raspberry-pi-3.html>>. Acesso em: 20 maio 2016.

TOTAL CAR. ELM327 Review & About ELM 327 OBD2 Interface. **Total Car Diagnostics Support**, [S.l.], p. 1, 2014. Disponível em: <<http://www.totalcardiagnostics.com/support/Knowledgebase/Article/View/72/15/elm327-review--about-elm-327-obd2-interface>>. Acesso em: 20 maio 2016.

W3SCHOOLS. HTTP Methods: GET vs. POST. **W3Schools**, p. 1, 2016. Disponível em: <http://www.w3schools.com/TAGs/ref_httpmethods.asp>. Acesso em: 20 ago. 2016.

ZAMBARDA, P. "Internet das Coisas": entenda o conceito e o que muda com a tecnologia. **Tech Tudo**, [S.l.], p. 1, 2014. Disponível em: <<http://www.techtudo.com.br/noticias/noticia/2014/08/internet-das-coisas-entenda-o-conceito-e-o-que-muda-com-tecnologia.html>>. Acesso em: 20 maio 2016.

ZURAWSKI, R. Automotive Embedded Systems Handbook. Boca Raton: CRC Press, 2009. p. 33-34. ISBN 978-0-8493-8026-6.