|  |
| --- |
| 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 auxilio extraclasse durante a realização deste trabalho.

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

Ao professor Roberto Heinzle, pelo entusiasmo com que sempre ministrou as aulas de estruturas de dados e pelo auxilio 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](#_Toc469147603)

[Figura 2 - Aspecto da interface ELM327 RS232 17](#_Toc469147604)

[Figura 3 - Aspecto da interface ELM327 USB 17](#_Toc469147605)

[Figura 4 - Aspecto da interface ELM327 Bluetooth 18](#_Toc469147606)

[Figura 5 - Aspecto da interface ELM327 WiFi 18](#_Toc469147607)

[Figura 6 - Blocos eletrônicos da interface ELM327 19](#_Toc469147608)

[Figura 7 - Visão geral dos protocolos de comunicação OBD 20](#_Toc469147609)

[Figura 8 - Características do Raspberry Pi 3 Model B 21](#_Toc469147610)

[Figura 9 - Conectando PyOBD com o veículo 23](#_Toc469147611)

[Figura 10 - Exibindo resultados de testes com PyOBD 23](#_Toc469147612)

[Figura 11 - Verificando dados em tempo real com PyOBD 24](#_Toc469147613)

[Figura 12 - Lendo e limpando códigos de falhas com PyOBD 24](#_Toc469147614)

[Figura 13 - Velocidade do veículo no EnviroCar 25](#_Toc469147615)

[Figura 14 - Velocidade média, trajeto e distância percorridos no EnviroCar 26](#_Toc469147616)

[Figura 15 - Informações coletadas pelo EnviroCar durante o percurso 26](#_Toc469147617)

[Figura 16 - Ciclo de vida do firmware 29](#_Toc469147618)

[Figura 17 - Ciclo de vida do servidor 30](#_Toc469147619)

[Figura 18 - Camadas e pacotes do firmware 31](#_Toc469147620)

[Figura 19 - Relacionamento entre as principais classes do firmware 34](#_Toc469147621)

[Figura 20 - Leitura de dados da interface ELM327 Bluetooth 36](#_Toc469147622)

[Figura 21 - Envio dos dados pendentes 37](#_Toc469147623)

[Figura 22 - Processamento de requisições no servidor 38](#_Toc469147624)

[Figura 23 - Relacionamento entre as classes do servidor 39](#_Toc469147625)

[Figura 24 - Versões do Sistema Operacional e Java no Raspberry Pi 40](#_Toc469147626)

[Figura 25 - Arquitetura da API JABWT BlueCove 41](#_Toc469147627)

[Figura 26 - Executando ObdJrpListDevices no Raspberry Pi 48](#_Toc469147628)

[Figura 27 - Executando ObdJrpScanData no Raspberry Pi 49](#_Toc469147629)

[Figura 28 - Interfaces ELM327 WiFi e Bluetooth 50](#_Toc469147630)

[Figura 29 - Instalação no Volkswagen Gol 2010 72](#_Toc469147631)

[Figura 30 - Instalação no Volkswagen SpaceFox 2009 72](#_Toc469147632)

[Figura 31 - Aspecto do adaptador USB veicular 73](#_Toc469147633)

[Figura 32 - Aspecto do mini modem USB 73](#_Toc469147634)

[Figura 33 - Página inicial 74](#_Toc469147635)

[Figura 34 - Página com leituras em tempo real 74](#_Toc469147636)

[Figura 35 - Página com leituras gráficas em tempo real 75](#_Toc469147637)

[Figura 36 - Página com histórico de leituras 75](#_Toc469147638)

LISTA DE Quadros

[Quadro 1 - Listando dispositivos e serviços Bluetooth 43](#_Toc469147639)

[Quadro 2 - Disparando consulta de dispositivos com JABWT 44](#_Toc469147640)

[Quadro 3 - DiscoveryListener para consulta de dispositivos 44](#_Toc469147641)

[Quadro 4 - Classe Lock utilizada para sincronização de processos 45](#_Toc469147642)

[Quadro 5 - Disparando consulta de serviços com JABWT 46](#_Toc469147643)

[Quadro 6 - DiscoveryListener para consulta de serviços 47](#_Toc469147644)

[Quadro 7 - Configuração de acesso Bluetooth 49](#_Toc469147645)

[Quadro 8 - Método execute da classe ObdJrpScanData 51](#_Toc469147646)

[Quadro 9 - Reestabelecendo conexão Bluetooth após erro 51](#_Toc469147647)

[Quadro 10 - Métodos restartAfterError e stop da classe ObdJrpScanData 52](#_Toc469147648)

[Quadro 11 - Atualizando interface de usuário em tempo real 52](#_Toc469147649)

[Quadro 12 - Apresentando dados lidos na interface de usuário 53](#_Toc469147650)

[Quadro 13 - Configuração dos Parsers 54](#_Toc469147651)

[Quadro 14 - Construtor da classe Scanner 54](#_Toc469147652)

[Quadro 15 - Inicializando objeto ELM327 55](#_Toc469147653)

[Quadro 16 - Obtendo PIDs suportados pelo veículo 56](#_Toc469147654)

[Quadro 17 - Processando máscara de bits 57](#_Toc469147655)

[Quadro 18 - Classe EventMulticaster 58](#_Toc469147656)

[Quadro 19 - Método execute da classe ScanLoop 59](#_Toc469147657)

[Quadro 20 - Construtor da classe ScanLoop 59](#_Toc469147658)

[Quadro 21 - Tratamento de eventos na classe ScanUploader 60](#_Toc469147659)

[Quadro 22 - Método upload da classe ScanUploader 61](#_Toc469147660)

[Quadro 23 - Método add da classe PackagePersister 62](#_Toc469147661)

[Quadro 24 - Método execute da classe DataMonitor 62](#_Toc469147662)

[Quadro 25 - Método upload da classe DataMonitor 63](#_Toc469147663)

[Quadro 26 - Classe PostRequest 64](#_Toc469147664)

[Quadro 27 - Classe ObdJrpServlet 65](#_Toc469147665)

[Quadro 28 - Command SendData 66](#_Toc469147666)

[Quadro 29 - Command UploadData 66](#_Toc469147667)

[Quadro 30 - Command ReadData 67](#_Toc469147668)

[Quadro 31 - Página vehicle-history.jsp 68](#_Toc469147669)

[Quadro 32 - Página vehicle-detail.jsp 69](#_Toc469147670)

[Quadro 33 - Command ViewChart 70](#_Toc469147671)

[Quadro 34 - Página view-chart.jsp 71](#_Toc469147672)

[Quadro 35 - Command ListVehicles 71](#_Toc469147673)

[Quadro 36 - PIDs suportados do Gol e SpaceFox 77](#_Toc469147674)

[Quadro 37 - Comparativo entre os trabalhos correlatos e o trabalho desenvolvido 78](#_Toc469147675)

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](#_Toc469147676)

[1.1 OBJETIVOS DO TRABALHO 11](#_Toc469147677)

[1.2 ESTRUTURA 11](#_Toc469147678)

[2 FUNDAMENTAÇÃO TEÓRICA 12](#_Toc469147679)

[2.1 HISTÓRIA DO OBD 12](#_Toc469147680)

[2.2 PROTOCOLOS OBD2 14](#_Toc469147681)

[2.3 MODOS DE DIAGNÓSTICO 15](#_Toc469147682)

[2.4 INTERFACE ELM327 16](#_Toc469147683)

[2.5 RASPBERRY PI 20](#_Toc469147684)

[2.6 TRABALHOS CORRELATOS 22](#_Toc469147685)

[2.6.1 PYOBD 22](#_Toc469147686)

[2.6.2 ENVIROCAR 25](#_Toc469147687)

[3 DESENVOLVIMENTO DO PROTÓTIPO 27](#_Toc469147688)

[3.1 REQUISITOS 27](#_Toc469147689)

[3.2 ESPECIFICAÇÃO 28](#_Toc469147690)

[3.2.1 ESPECIFICAÇÃO DO FIRMWARE 30](#_Toc469147691)

[3.2.2 ESPECIFICAÇÃO DO SERVIDOR 37](#_Toc469147692)

[3.3 IMPLEMENTAÇÃO 40](#_Toc469147693)

[3.3.1 Técnicas e ferramentas utilizadas 40](#_Toc469147694)

[3.3.2 Operacionalidade da implementação 72](#_Toc469147695)

[3.4 RESULTADOS e DISCUSSÕES 76](#_Toc469147696)

[4 CONCLUSÕES 79](#_Toc469147697)

[4.1 EXTENSÕES 79](#_Toc469147698)

[Referências 80](#_Toc469147699)

# 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.

## 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:

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

## 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.

# FUNDAMENTAÇÃO TEÓRICA

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

## 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:

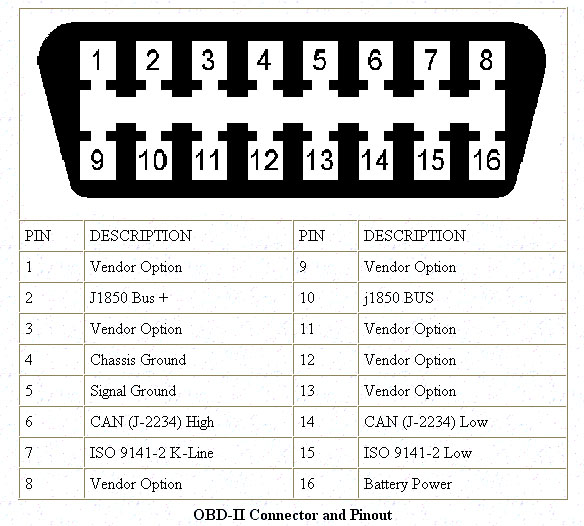
1. sensor de pressão absoluta ou fluxo de ar;
2. sensor de posição da borboleta;
3. sensor de temperatura de arrefecimento;
4. sensor de temperatura de ar;
5. sensor de oxigênio;
6. sensor de velocidade do veículo;
7. sensor de posição do eixo comando de válvulas;
8. sensor de posição do virabrequim;
9. sistemas de recirculação dos gases de escape;
10. sensor para detecção de detonação;
11. válvulas injetoras;
12. sistema de ignição;
13. módulo controle eletrônico do motor;
14. lâmpada indicadora de mau funcionamento;
15. 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:

1. sensores de oxigênio (pré e pós-catalisador);
2. eletroválvula do cânister;
3. 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.

## 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:

1. ISO 157565: Utilizado por todos os veículos. Velocidade de comunicação de 125 a 500 Kbps;
2. 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:

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

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

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

## MODOS DE DIAGNÓSTICO

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

1. 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;
2. modo 2: Obtém o “instantâneo” de uma falha. Quando a ECU detecta uma falha, ela grava os dados do sensor daquele momento específico;
3. 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:
   1. P0xxx: Para falhas associadas ao motor e transmissão;
   2. C0xxx: Para falhas associadas ao chassi;
   3. B0xxx: para falhas associadas à carroceria;
   4. U0xxx: para falhas associadas à comunicação de rede.
4. modo 4: Utilizado para apagar os DTCs gravados e desligar o MIL;
5. modo 5: Retorna o autodiagnostico do sensor lambda[[1]](#footnote-1). Segundo Outils OBD Facile (2015), este modo não é mais utilizado pois o modo 6 substitui suas funções;
6. modo 6: Retorna os resultados do autodiagnostico realizado nos diversos sensores do veículo;
7. 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;
8. 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;
9. modo 9: Este modo obtém informações do veículo como por exemplo seu número de identificação;
10. 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).

## 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:

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



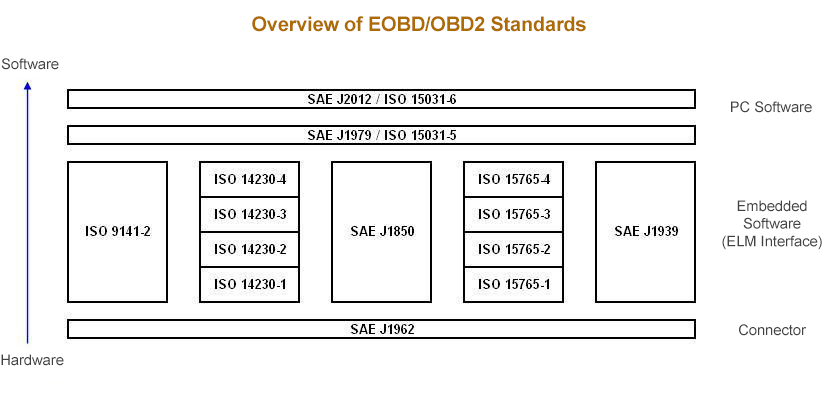
Fonte: Total Car (2014).

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

1. 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;
2. 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;
3. 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



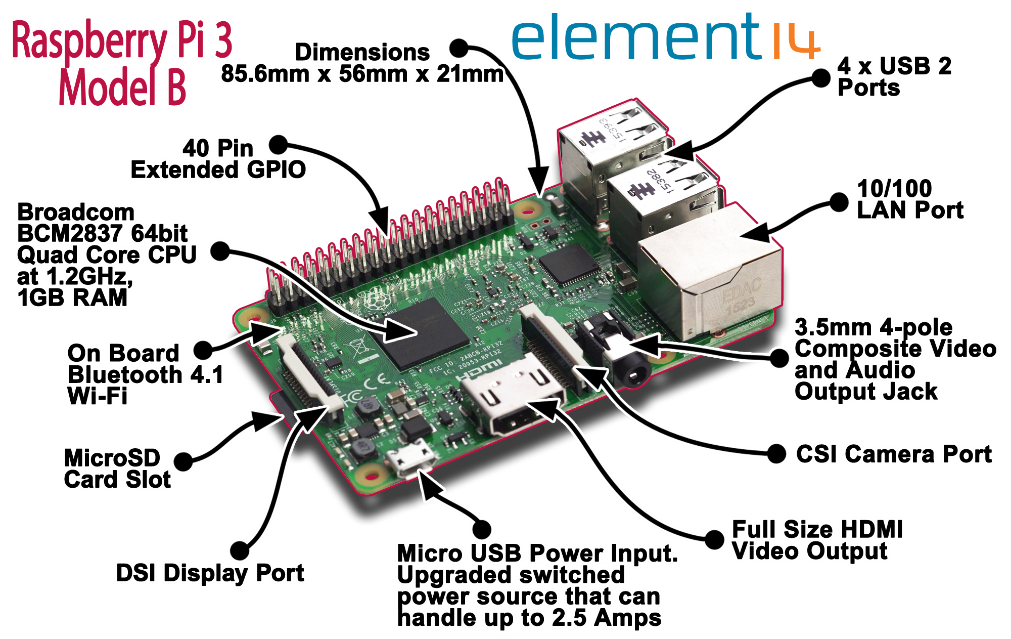
Fonte: Total Car (2014).

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.

## 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:

1. computador de placa única com chipset Broadcom BCM2837;
2. processador quad core ARM Cortex-A53 de 1,2GHz;
3. 1GB de Random Access Memory (RAM);
4. 40 pinos de General Purpose Input/Output (GPIO);
5. conexão Bluetooth 4.1 integrada;
6. conexão WiFi 802.11n integrada;
7. 1 porta Ethernet 10/100;
8. 4 portas USB;
9. 1 conector de 4 polos, combinado para saída de áudio estéreo e vídeo composto;
10. 1 saída High Definition Multimedia Interface (HDMI);
11. 1 porta Camera Serial Interface (CSI);
12. 1 porta Display Serial Interface (DSI);
13. 1 porta micro SanDisk (SD), para carga do sistema operacional e armazenamento de dados;
14. 1 porta micro USB para fonte de alimentação.

## 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).

### 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:

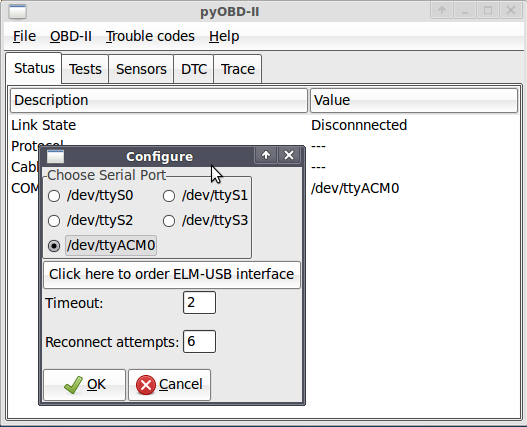
1. uma interface ELM327 USB;
2. python 2.x ou superior;
3. pacote py\_serial;
4. um veículo que implemente o padrão OBD2.

Com o PyOBD é possível:

1. conectar-se ao veículo;
2. exibir resultados de testes;
3. verificar dados dos sensores em tempo real;
4. 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.

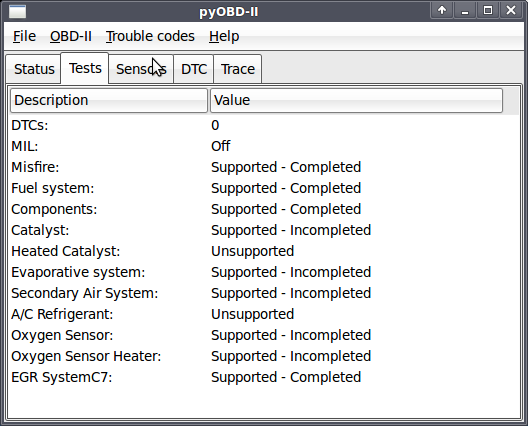
Figura 9 - Conectando PyOBD com o veículo



Fonte: PyOBD (2015).

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

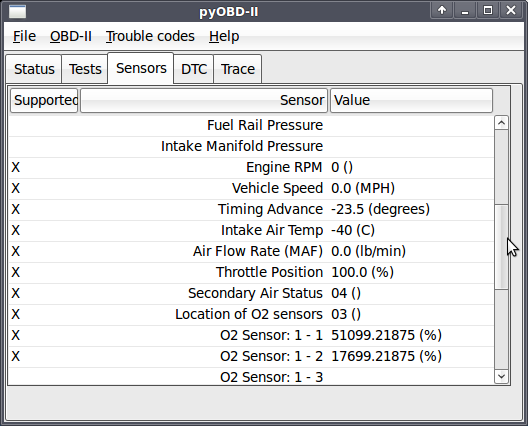
Figura 10 - Exibindo resultados de testes com PyOBD



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.

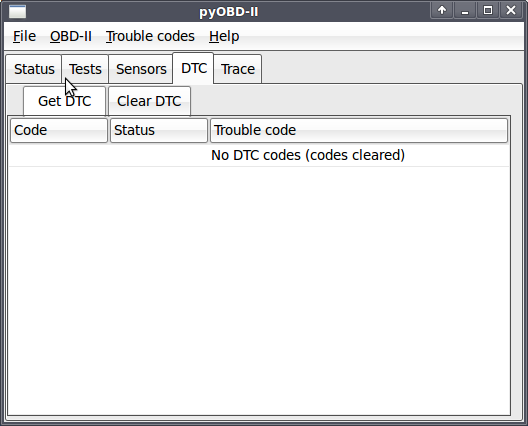
Figura 11 - Verificando dados em tempo real com PyOBD



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).

### 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 carbónico 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.

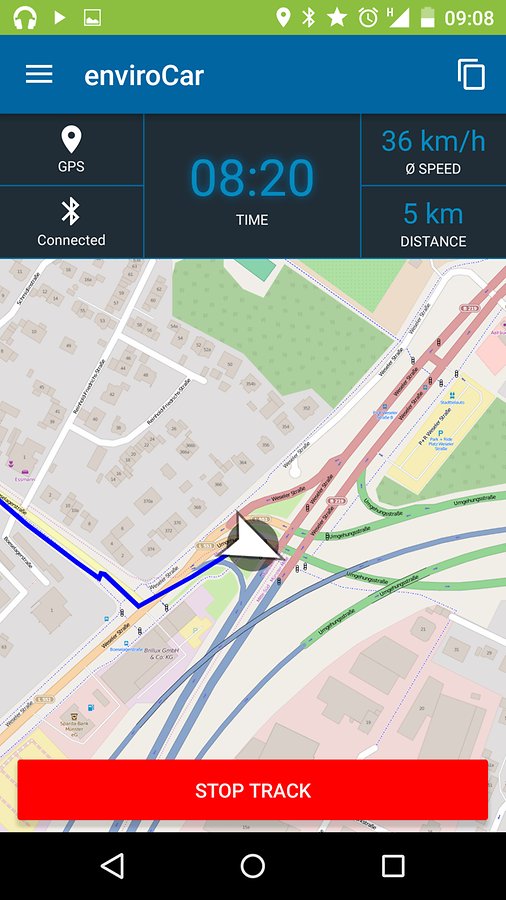
Figura 13 - Velocidade do veículo no EnviroCar



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.

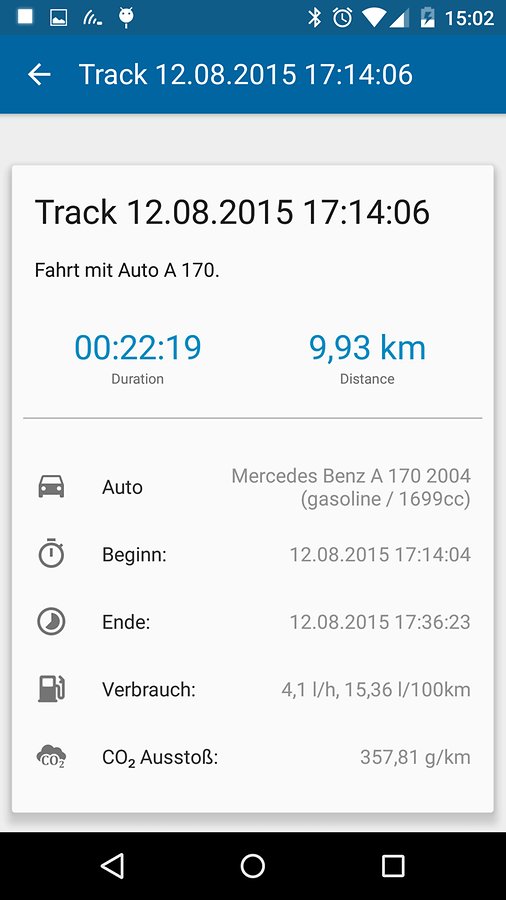
Figura 14 - Velocidade média, trajeto e distância percorridos no EnviroCar



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).

# 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.

## 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:

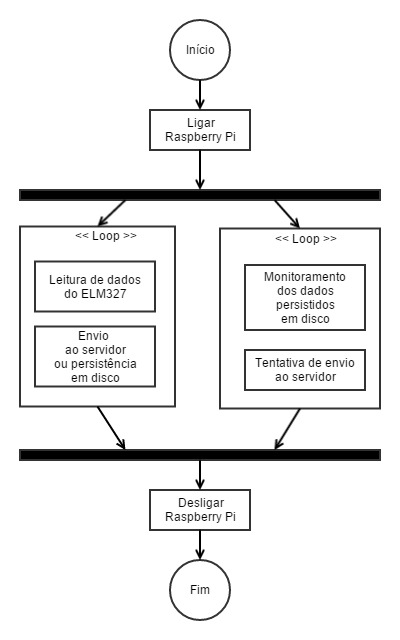
1. o firmware deve ser inicializado automaticamente ao ligar a placa Raspberry Pi (Requisito Funcional – RF);
2. o firmware deve se conectar à porta OBD2 através de uma interface ELM327 Bluetooth (RF);
3. o firmware deve coletar os dados da porta OBD2 e armazená-los localmente até serem enviados ao servidor (RF);
4. o firmware deve tentar estabelecer uma conexão com o servidor a cada 5 minutos, caso não esteja conectado à internet (Requisito Não Funcional – RNF);
5. 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);
6. o firmware deve ser desenvolvido utilizando tecnologia Java SE (RNF);
7. o firmware deve executar em sistema operacional Raspbian (RNF);
8. o servidor deve responder requisições HTTP, através dos métodos GET e POST[[2]](#footnote-2) (RF);
9. o servidor deve persistir os dados coletados pelo firmware (RF);
10. o servidor deve persistir os dados em arquivos XML, sem a necessidade de utilizar banco de dados (RNF);
11. o servidor deve dispor uma página web para consultar os dados OBD2 a partir do número do chassi do carro (RF);
12. o servidor deve ser desenvolvido utilizando tecnologia Java EE (RNF);
13. o servidor deve executar no servidor de aplicações Apache TomCat (RNF);
14. a página web deve apresentar gráficos com os valores dos dados coletados (RF);
15. a página web deve apresentar uma tabela com os valores dos dados coletados (RF);
16. a página web deve ter interface responsiva de modo que possa ser visualizada em smartphones (RNF);
17. a página web deve ser desenvolvida utilizando HTML, CSS e JavaScript (RNF).

## 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

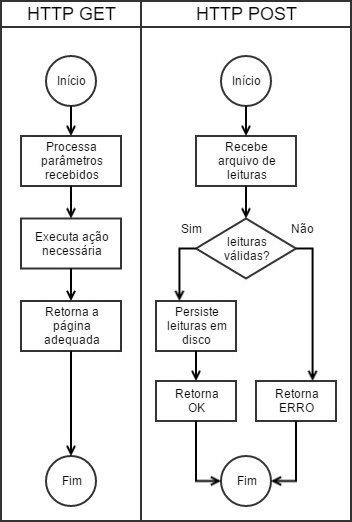


Fonte: Elaborado pelo autor.

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



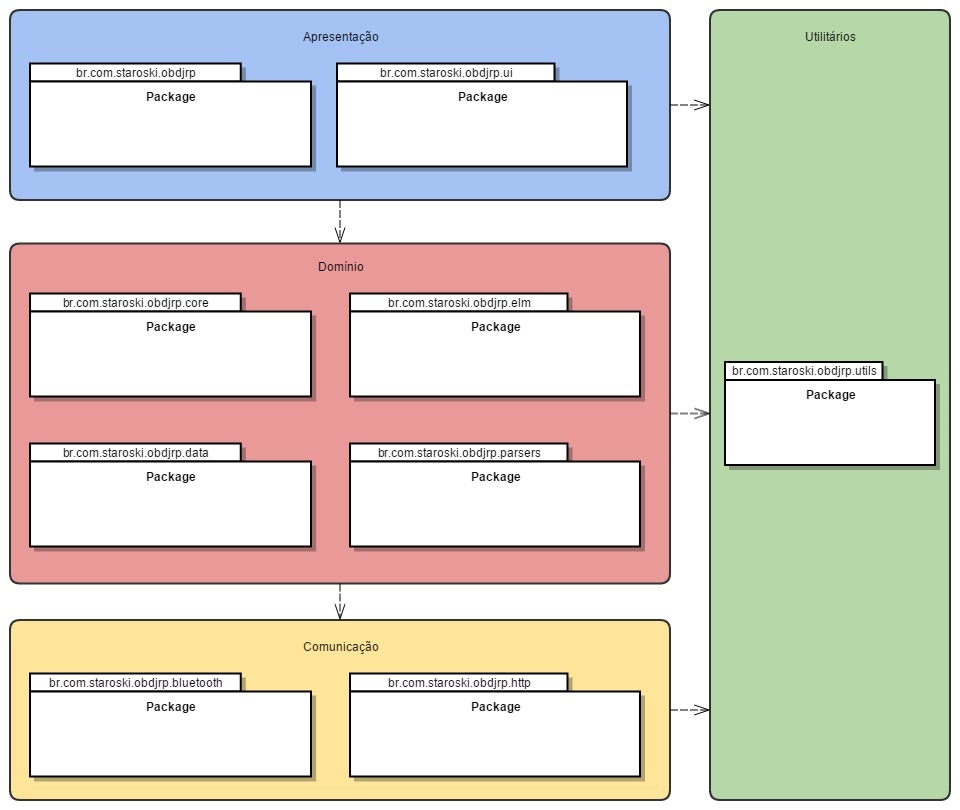
Fonte: Elaborado pelo autor.

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.

### 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.obdjrp 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:

1. ObdJrpListDevices, programa que lista os dispositivos Bluetooth e seus serviços disponíveis;
2. ObdJrpScanData, programa que lê os dados da ECU através de uma interface ELM327 Bluetooth, tentando enviá-los ao servidor ou persistindo-os em disco;
3. 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:

1. Config, classe que implementa o padrão de projeto Singleton[[3]](#footnote-3) e representa a configuração dos programas, que é realizada através de um arquivo texto chamado obd-jrp.properties;
2. 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;
3. 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;
4. Scanner, classe que realiza a leitura dos dados da ECU, comunicando-se com a interface ELM327, esta classe implementa o padrão de projeto Observer[[4]](#footnote-4), sendo possível registrar objetos para serem notificados quando as leituras são concluídas ou quando ocorrem erros;
5. 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 harware conectado ao veículo. A classe ELM327Error representa um erro que pode ser lançado pela classe ELM327. Neste pacote existe ainda uma classe não pública chamada Disconnector, que se registra à máquina virtual Java através do método java.lang.Runtime.addShutdowHook e quando a máquina virtual Java é encerrada, o Disconnector itera sobre a lista de objetos ELM327 ativos e os desconecta.

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

1. 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;
2. Scan, esta classe representa um objeto de leitura, que contém uma lista com os objetos Data lidos durante um ciclo de leitura;
3. 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;
4. Parser, é uma interface Strategy[[5]](#footnote-5) para objetos que implementam algoritmos que convertem objetos do tipo Data em um objetos do tipo Parsed;
5. 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;
6. Parsing, classe utilitária que utiliza implementações de Parser para transformar objetos Data em objetos Parsed.

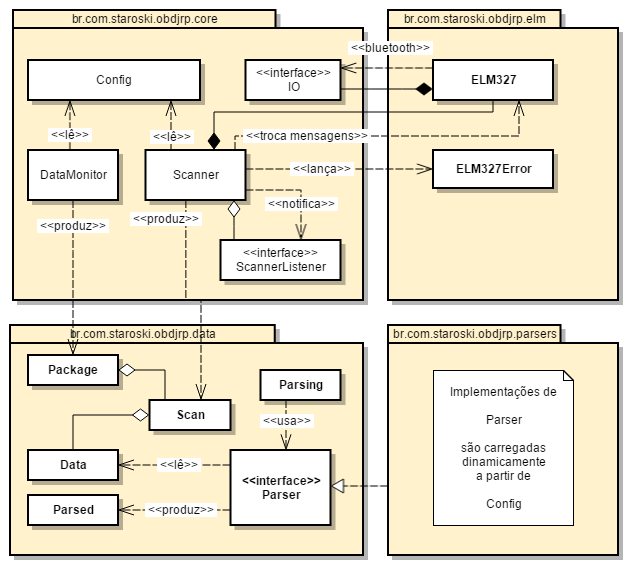
O pacote br.com.staroski.obdjrp.parsers define algumas classes que implementam a interface Parser. Como supracitado, um objeto Parser é responsável por converter um objeto Data em um objeto Parsed. Considerando o cenário hipotético de um objeto Data, com as propriedades pid=0C e value=0AF0, este objeto corresponde a uma leitura das Rotações por Minuto (RPM)[[6]](#footnote-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 RPM[[7]](#footnote-7), 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.obdjrp.bluetooth define a classe Bluetooth, responsável por simplificar o acesso à API JABWT responsável pela descoberta de dispositivos Bluetooth e conexão aos mesmos. A classe Bluetooth implementa o padrão de projeto Façade[[8]](#footnote-8) 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.obdjrp.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



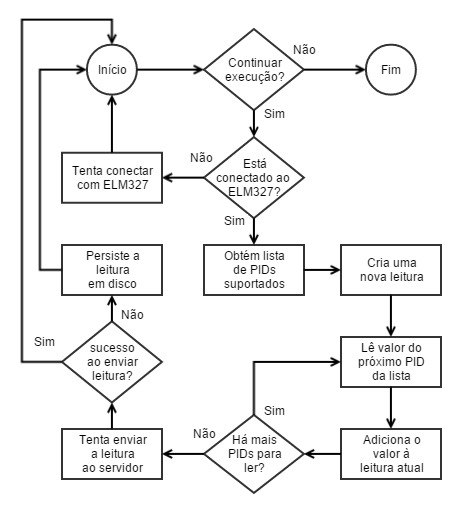
Fonte: Elaborado pelo autor.

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.

#### 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.

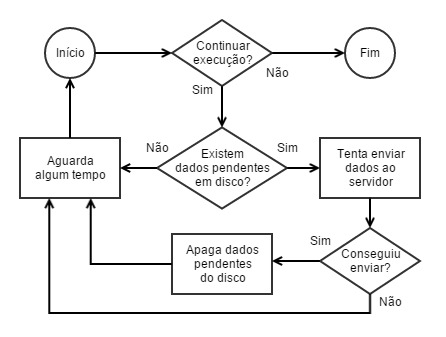
Na Figura 20 é apresentado o algoritmo de leitura da interface ELM327 Bluetooth, que consiste em um laço de repetição que realiza as seguintes operações:

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

#### 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.

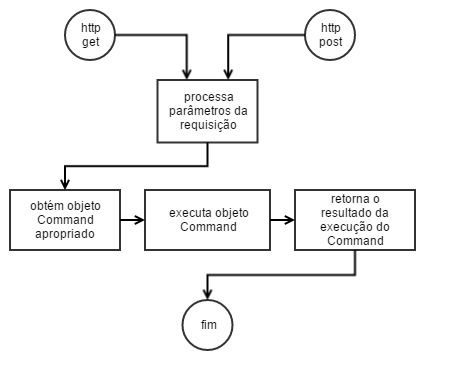
Assim como a leitura dos dados da interface ELM327 Bluetooth, o envio dos dados pendentes também consiste em um laço de repetição que realiza as seguintes operações:

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

### 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 Command[[9]](#footnote-9), 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



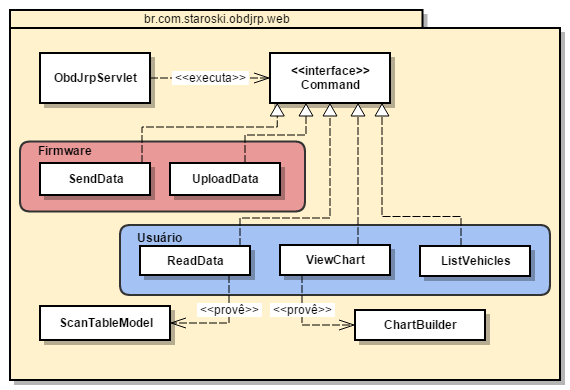
Fonte: Elaborado pelo autor.

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

1. 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;
2. Command, interface para os objetos que tratam parâmetros específicos das requisições recebidas pelo ObdJrpServlet;
3. SendData, implementação de Command invocada pelo firmware para enviar leituras em tempo real;
4. UploadData, implementação de Command invocada pelo firmware para enviar pacotes de leituras pendentes;
5. ListVehicles, implementação de Command invocada pelo usuário ao abrir a página com a lista de veículos;
6. ReadData, implementação de Command invocada pelo usuário ao abrir a página para visualizar leituras em tempo real;
7. ViewChart, implementação de Command invocada pelo usuário ao selecionar uma leitura para visualização gráfica;
8. ScanTableModel, provê métodos para a página renderizar uma tabela com as leituras;
9. 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.

## 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.

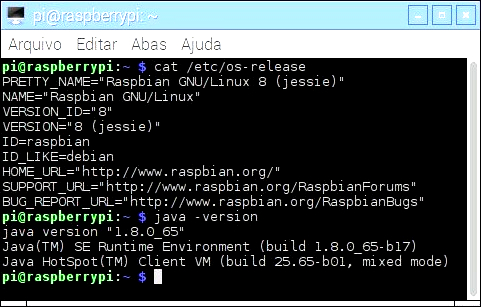
### Técnicas e ferramentas utilizadas

As implementações tanto do firmware quanto do servidor, foram realizadas utilizado 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.

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

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

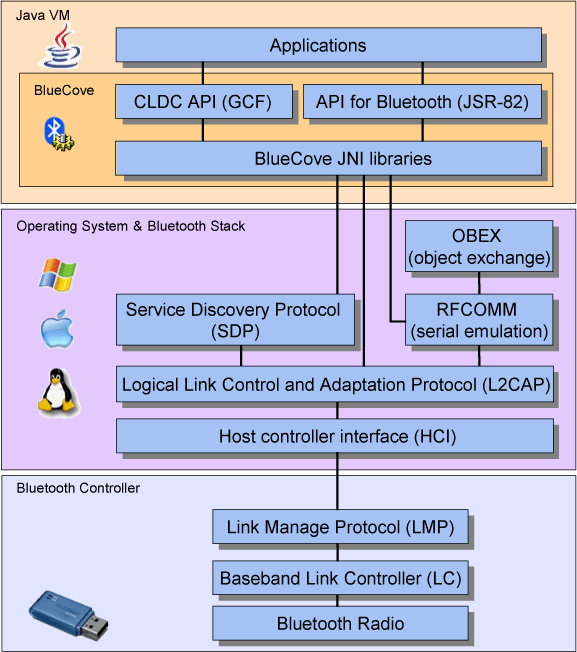
Figura 24 - Versões do Sistema Operacional e Java no Raspberry Pi



Fonte: Elaborado pelo autor.

Conforme citado na seção 2.4, 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 repositório[[10]](#footnote-10) 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 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:

1. instalar as ferramentas Maven e Ant no Raspberry Pi;
2. executar o comando Maven para criar os diretórios de código fonte compatíveis com o ambiente Eclipse:
   1. mvn eclipse:clean eclipse:eclipse -DdownloadSources=true;
3. executar o comando Ant para compilar as bibliotecas nativas e gerar o arquivo JAR da biblioteca Java:
   1. ant all.

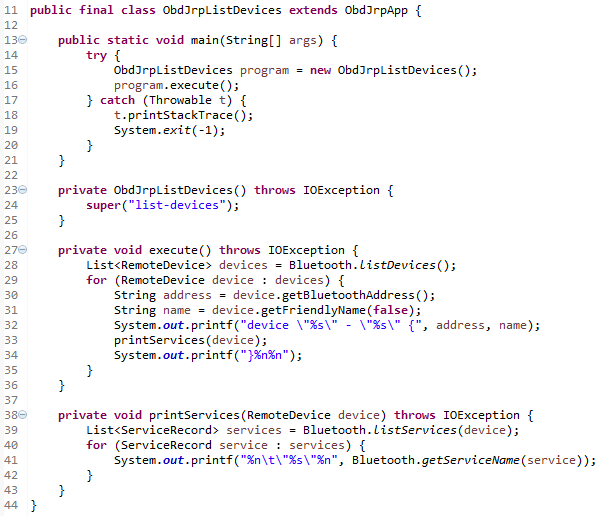
Com a biblioteca BlueCove compilada para arquitetura ARM, foi possível realizar no próprio Raspberry algumas provas de conceito para avaliar se seria viável dar continuidade ao desenvolvimento utilizando a linguagem Java. Os testes consistiram em listar os dispositivos Bluetooth pareados e tentar obter 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 Python[[11]](#footnote-11), 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.

#### 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 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

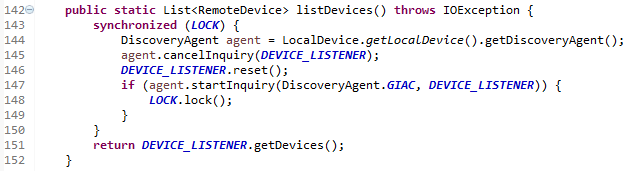


Fonte: Elaborado pelo autor.

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.

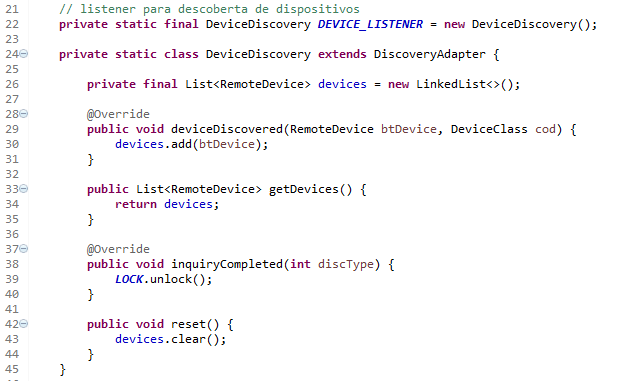
Quadro 2 - Disparando consulta de dispositivos com JABWT



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

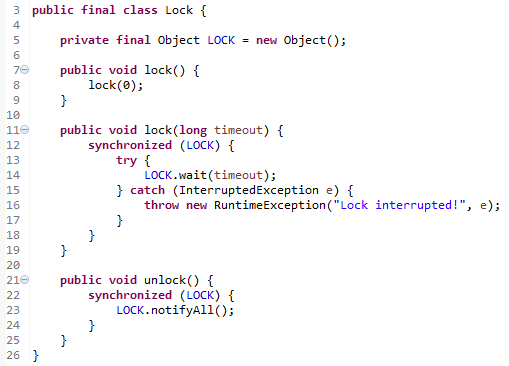


Fonte: Elaborado pelo autor.

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.

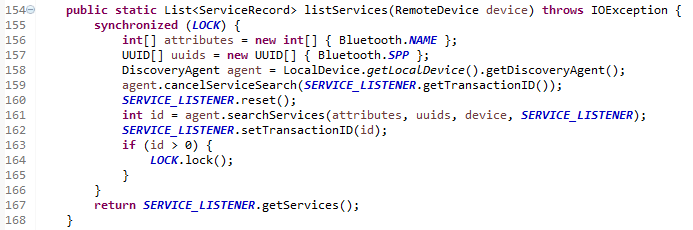
Quadro 4 - Classe Lock utilizada para sincronização de processos



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 documentação[[12]](#footnote-12) 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.

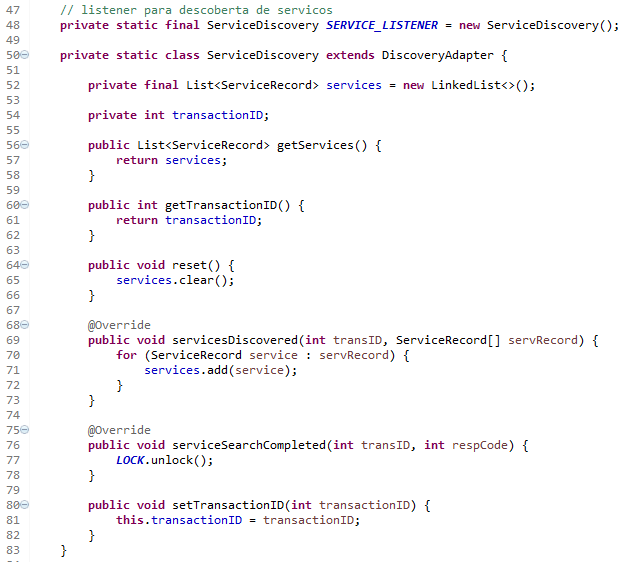
Quadro 5 - Disparando consulta de serviços com JABWT



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.

Quadro 6 - DiscoveryListener para consulta de serviços

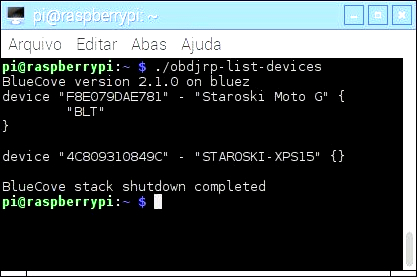


Fonte: Elaborado pelo autor.

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

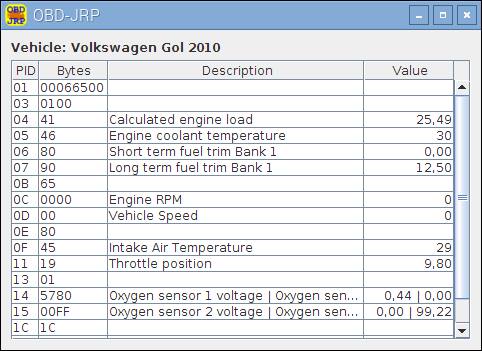


Fonte: Elaborado pelo autor.

#### 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 removo através do VNC[[13]](#footnote-13). Na Figura 27 observa-se o aspecto da janela apresentada pelo ObdJrpScanData.

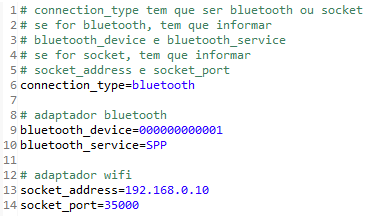
Figura 27 - Executando ObdJrpScanData no Raspberry Pi



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



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 instancias 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

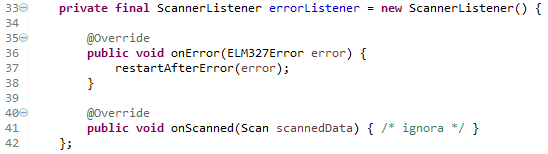


Fonte: Elaborado pelo autor.

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

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

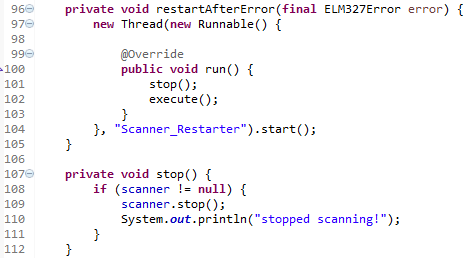
Quadro 9 - Reestabelecendo conexão Bluetooth após erro



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.

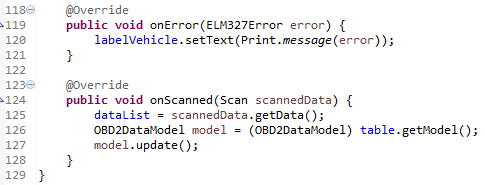
Quadro 10 - Métodos restartAfterError e stop da classe ObdJrpScanData



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.

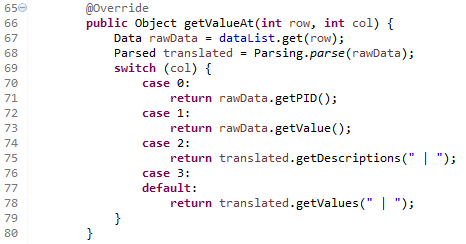
Quadro 11 - Atualizando interface de usuário em tempo real



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]](#footnote-14) no qual parte relevante da implementação é apresentada no Quadro 12.

Quadro 12 - Apresentando dados lidos na interface de usuário

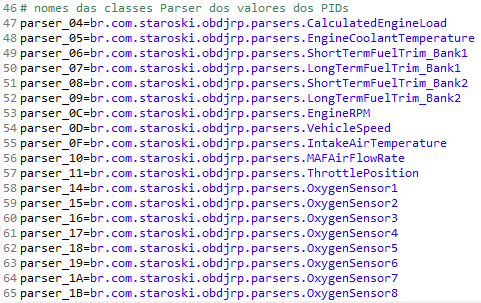


Fonte: Elaborado pelo autor.

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]](#footnote-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 presentar 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.

Quadro 13 - Configuração dos Parsers

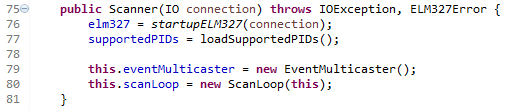


Fonte: Elaborado pelo autor.

#### Comunicação com a interface ELM327 Bluetooth

Para efetivamente trocar mensagens com a interface ELM327, a classe Scanner utiliza uma instância da classe ELM37. 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.

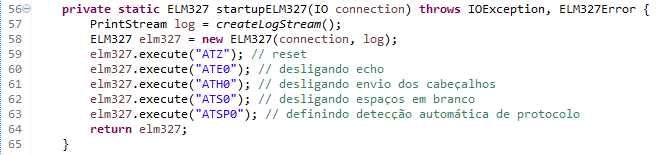
Quadro 14 - Construtor da classe Scanner



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 permance num laço contínuo executando as leituras do ELM327. No Quadro 19 observa-se a implementação do método execute da classe ScanLoop.

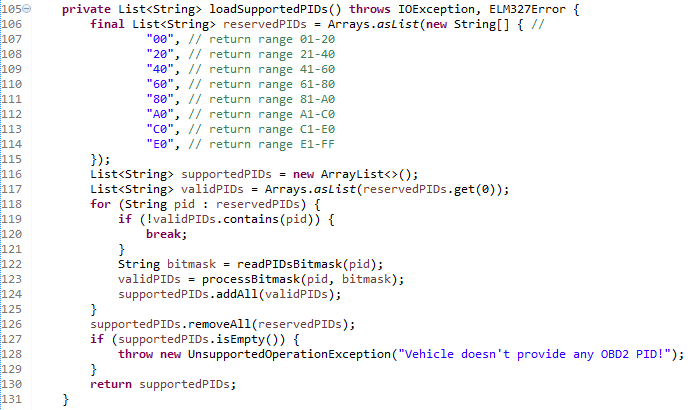
Quadro 15 - Inicializando objeto ELM327



Fonte: Elaborado pelo autor.

Na linha 59 é executado o comando[[16]](#footnote-16) 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 estar configurações o objeto instanciado é retornado.

Quadro 16 - Obtendo PIDs suportados pelo veículo



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:

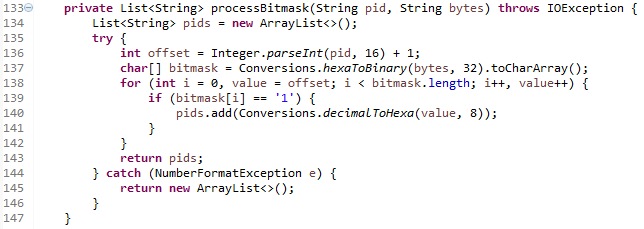
1. 00, obtém os PIDs válidos entre 01 e 20;
2. 20, obtém os PIDs válidos entre 21 e 40;
3. 40, obtém os PIDs válidos entre 41 e 60;
4. 60, obtém os PIDs válidos entre 61 e 80;
5. 80, obtém os PIDs válidos entre 81 e A0;
6. A0, obtém os PIDs válidos entre A1 e C0;
7. C0, obtém os PIDs válidos entre C1 e E0;
8. E0, obtém os PIDs válidos entre E1 e FF.

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

1. executa-se o PID 00;
2. obtém como resposta os bytes BEBACAFE;
3. convertendo binário, obtém-se
4. nesta máscara, o bit mais significativo tem peso 1 e o menos significativo, 32;
5. itera-se sobre os bits da esquerda para a direita, armazenando o peso de cada bit 1;
6. 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;
7. soma-se cada valor com o PID reservado, em decimal, que neste exemplo é 0;
8. 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.

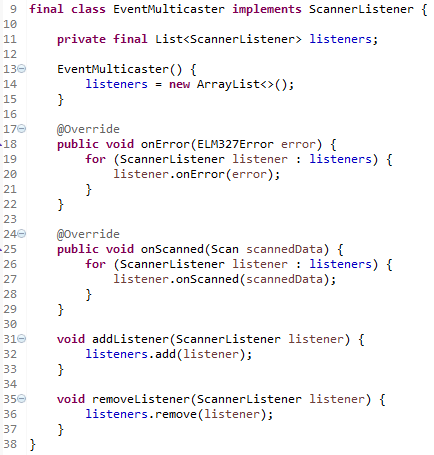
O algoritmo ilustrado no exemplo acima, é implementado pelo método processBitmask, apresentado no Quadro 17.

Quadro 17 - Processando máscara de bits



Fonte: Elaborado pelo autor.

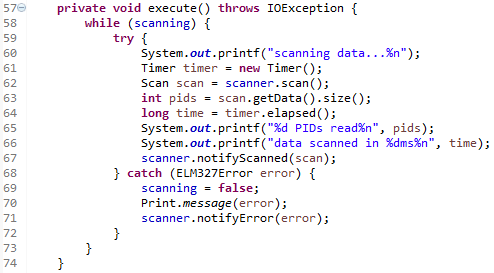
Quadro 18 - Classe EventMulticaster



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 Composite[[17]](#footnote-17).

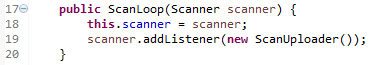
Quadro 19 - Método execute da classe ScanLoop



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.

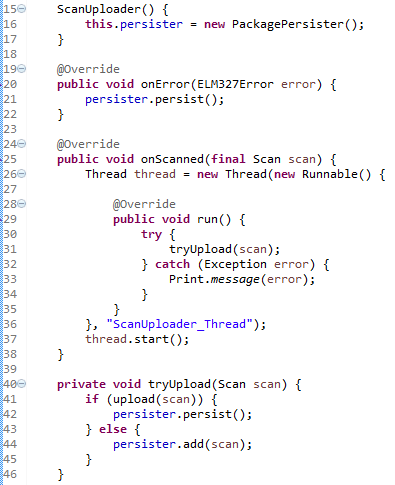
Quadro 20 - Construtor da classe ScanLoop



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 apresentado 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.

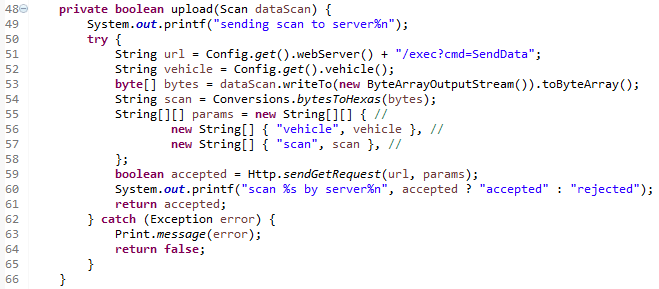
Quadro 21 - Tratamento de eventos na classe ScanUploader



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



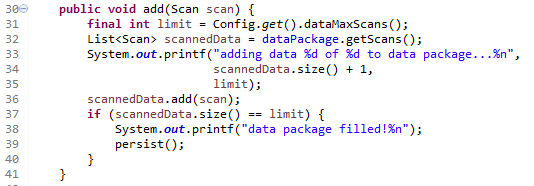
Fonte: Elaborado pelo autor.

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

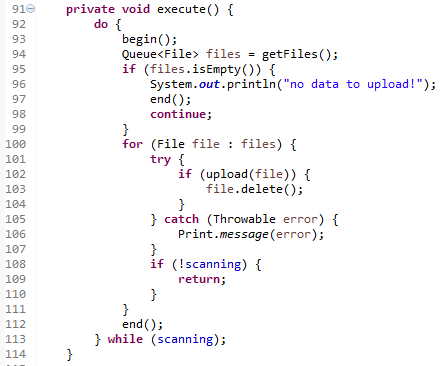


Fonte: Elaborado pelo autor.

#### 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.

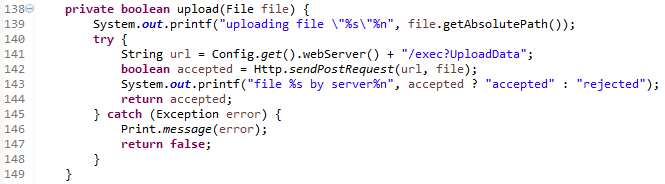
Quadro 24 - Método execute da classe DataMonitor



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



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



Fonte: Elaborado pelo autor.

#### 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

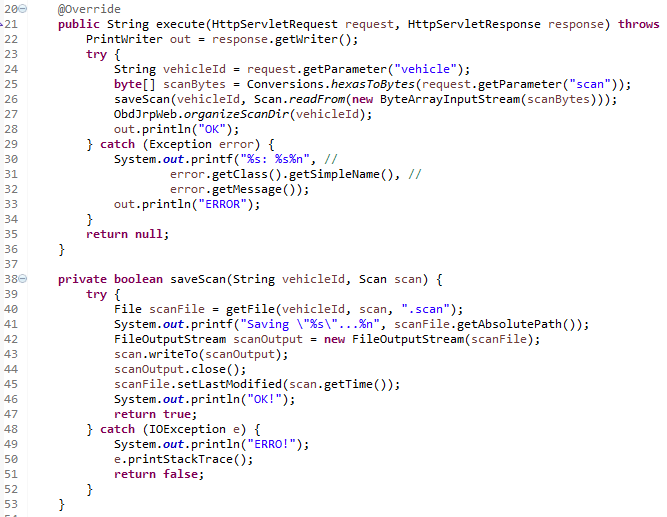


Fonte: Elaborado pelo autor.

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



Fonte: Elaborado pelo autor.

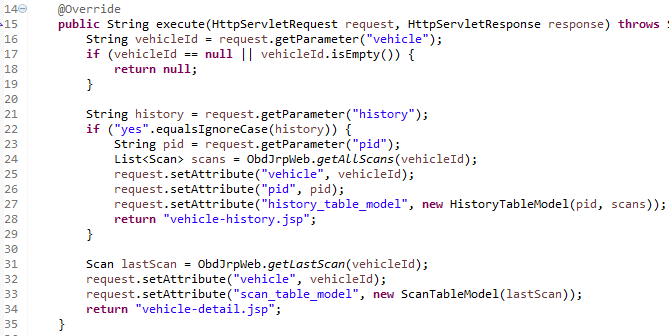
Quadro 29 - Command UploadData



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.

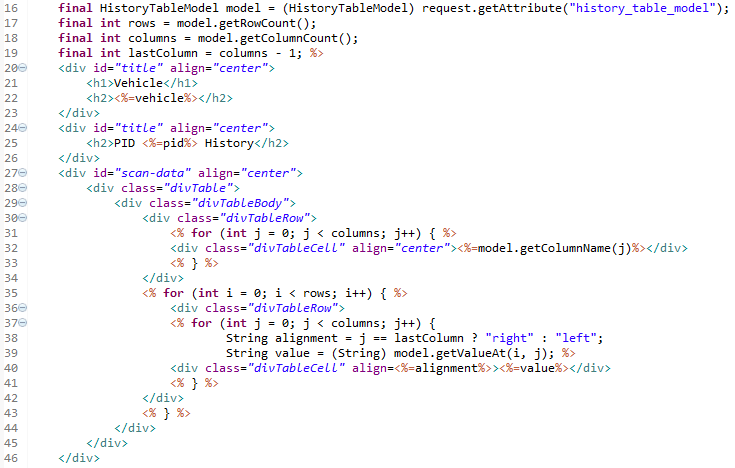
Quadro 30 - Command ReadData



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



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.

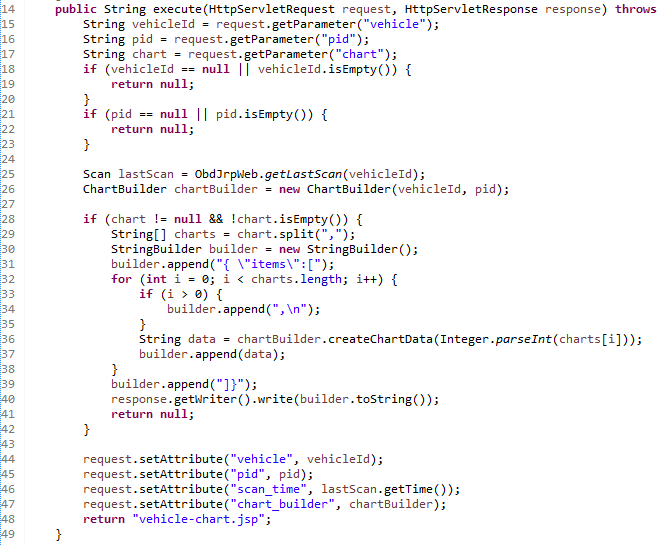
Quadro 32 - Página vehicle-detail.jsp



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



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

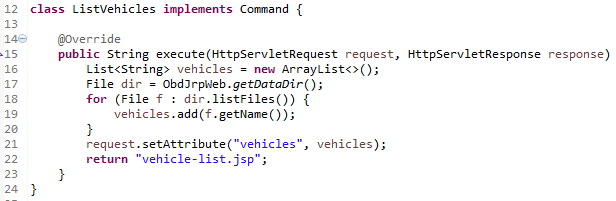


Fonte: Elaborado pelo autor.

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



Fonte: Elaborado pelo autor.

### 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.

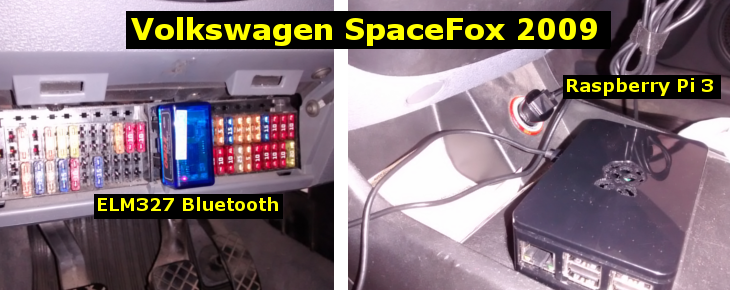
Figura 29 - Instalação no Volkswagen Gol 2010



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 web[[18]](#footnote-18) e visualiza a página inicial, apresentada na Figura 33. Nesta página ele informa o chassi[[19]](#footnote-19) do veículo e seleciona o botão “View”.

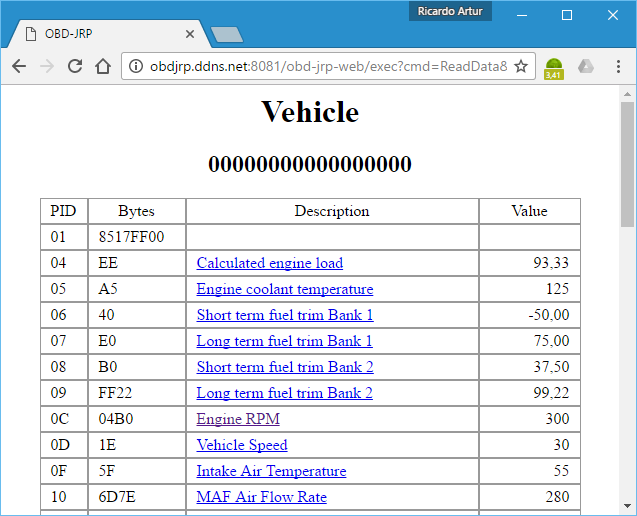
Figura 33 - Página inicial



Fonte: Elaborado pelo autor.

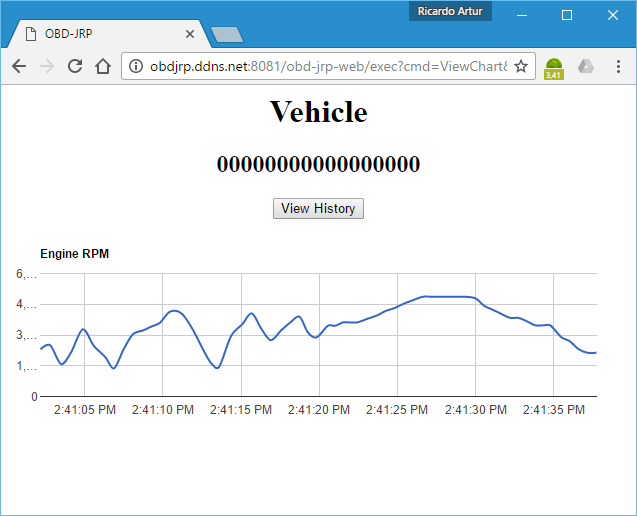
Após informar o chassi do veículo, o usuário é redirecionado para a página que mostra as leituras em tempo real, como pode ser observado na Figura 34. Nesta página, o usuário pode selecionar os links[[20]](#footnote-20) dos PIDs, de forma a obter uma visualização gráfica atualizada em tempo real, como pode ser observado na Figura 35.

Figura 34 - Página com leituras em tempo real



Fonte: Elaborado pelo autor.

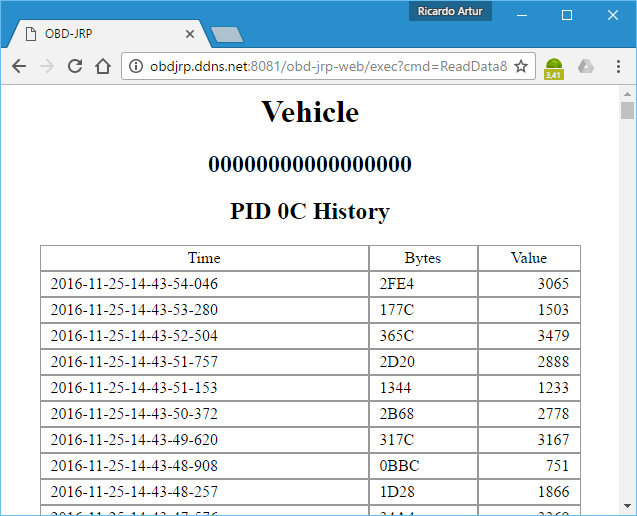
Figura 35 - Página com leituras gráficas em tempo real



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



Fonte: Elaborado pelo autor.

## RESULTADOS e DISCUSSÕES

O tempo de resposta entre o envio dos dados do firmware e a atualização das páginas no servidor se mostrou aceitável[[21]](#footnote-21). 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.

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.

# 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:

1. dimensões reduzidas;
2. fácil instalação em qualquer veículo que implemente OBD2;
3. independência de interação humana após instalado;
4. permite o monitoramento a partir de qualquer computador ou dispositivo móvel com acesso à internet.

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

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

## EXTENSÕES

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

1. integrar o protótipo com um banco de dados que comporte alto volume de dados;
2. integrar o protótipo com um sistema cognitivo que possa diagnosticar antecipadamente possíveis avarias ou manutenções a partir do histórico dos PIDs;
3. desenvolver hardware para desligar o protótipo quando o veículo é desligado ou quando o usuário desejar;
4. prover uma interface de usuário no próprio dispositivo, transformando-o em um computador de bordo;
5. 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. Disponivel 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. Disponivel em: <https://www.androidpit.com/app/org.envirocar.app>. Acesso em: 20 maio 2016.

BLUECOVE. **BlueCove**, p. 1, 2008. Disponivel em: <http://www.bluecove.org>. Acesso em: 20 out. 2016.

BLUETOOTH. Specification of the Bluetooth System. **Bluetooth**, p. 1907-1967, 2014. Disponivel 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. Disponivel 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. Disponivel em: <http://www.elmelectronics.com/DSheets/ELM327DS.pdf>. Acesso em: 20 maio 2016.

ENVIROCAR. Off we go. **EnviroCar**, [S.l.], p. 1, 2015. Disponivel em: <http://envirocar.org>. Acesso em: 20 maio 2016.

GSM ASSOCIATION. Understanding the Internet of Things (IoT). **GSM Association**, [S.l.], p. 1, 2014. Disponivel 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. Disponivel 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. Disponivel 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. Disponivel 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. Disponivel 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. Disponivel em: <http://www.outilsobdfacile.com/obd-mode-pid.php>. Acesso em: 20 maio 2016.

PYOBD. Open Source OBD2 Diagnostics. **OBD Tester**, [S.l.], p. 1, 2015. Disponivel 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. Disponivel em: <https://www.raspberrypi.org>. Acesso em: 20 maio 2016.

RIORAND. On Board Diagnostics. **RioRand Advanced Technology**, [S.l.], p. 1, 2015. Disponivel 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. Disponivel 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. Disponivel 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. Disponivel 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. Disponivel 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. Disponivel 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. Disponivel 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. Disponivel 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. Disponivel 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. Automative Embedded Systems Handbook. Boca Raton: CRC Press, 2009. p. 33-34. ISBN 978-0-8493-8026-6.

1. Horta (2000) explica que o sensor lambda é responsável pelo ajuste fino da mistura ar-combustível. [↑](#footnote-ref-1)
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. [↑](#footnote-ref-2)
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. [↑](#footnote-ref-3)
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. [↑](#footnote-ref-4)
5. Conforme Rocha (2005), Strategy permite que algoritmos mudem independentemente entre clientes que os utilizam. [↑](#footnote-ref-5)
6. SAE International (2006, p. 118-190) define os PIDs e respectivos algoritmos para obter informação legível. [↑](#footnote-ref-6)
7. Conforme SAE International (2006, p. 129), obtém-se as RPM, dividindo valor decimal dos bytes por 4. [↑](#footnote-ref-7)
8. Rocha (2005) explica que o padrão Façade define uma interface de nível mais elevado que torna o subsistema mais fácil de usar. [↑](#footnote-ref-8)
9. Segundo Rocha (2005), o padrão Command encapsula uma requisição na forma de um objeto, permitindo que clientes parametrizem diferentes requisições. [↑](#footnote-ref-9)
10. Repositório do BlueCove: http://repo1.maven.org/maven2/net/sf/bluecove/bluecove. [↑](#footnote-ref-10)
11. A API do trabalho correlato PyOBD poderia ser utilizado para o desenvolvimento em Python. [↑](#footnote-ref-11)
12. Documentação da classe UUID: http://www.bluecove.org/bluecove/apidocs/javax/bluetooth/UUID.html [↑](#footnote-ref-12)
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. [↑](#footnote-ref-13)
14. Documentação disponível em https://docs.oracle.com/javase/8/docs/api/javax/swing/table/TableModel.html. [↑](#footnote-ref-14)
15. Documentação disponível em https://docs.oracle.com/javase/8/docs/api/javax/swing/JTable.html. [↑](#footnote-ref-15)
16. Documentação disponível em: https://www.sparkfun.com/datasheets/Widgets/ELM327\_AT\_Commands.pdf. [↑](#footnote-ref-16)
17. Conforme Rocha (2005), Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme. [↑](#footnote-ref-17)
18. Para disponibilizar o acesso ao servidor, utilizou-se o serviço No-Ip, disponível em: https://www.noip.com. [↑](#footnote-ref-18)
19. Para os testes realizados, não foi utilizado o número real do chassi do veículo, ao invés disto, utilizou-se dígitos repetidos como 00000000000000000, 11111111111111111 e 222222222222222222. [↑](#footnote-ref-19)
20. Os links são renderizados somente para os PIDs dos quais foi implementado um Parser para obtenção de informação humanamente legível. [↑](#footnote-ref-20)
21. Vídeos dos testes disponibilizados em: https://goo.gl/b3GmGd. [↑](#footnote-ref-21)