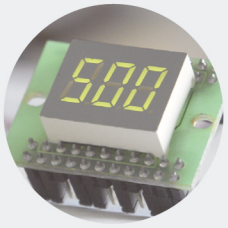
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES






SISTEMA DE LOCALIZAÇÃO DE VEÍCULOS PARA SMARTPHONE ANDROID

JOSÉ AFONSO LARANJEIRA PINA
Outubro de 2015

POLITÉCNICO
DO PORTO

SISTEMA DE LOCALIZAÇÃO DE VEÍCULOS PARA *SMARTPHONE ANDROID*

José Afonso Laranjeira Pina

Departamento de Engenharia Eletrotécnica

Instituto Superior de Engenharia do Porto

2015

Este relatório satisfaz, parcialmente, os requisitos que constam da Ficha de Disciplina de Tese/Dissertação, do 2º ano, do Mestrado em Engenharia Eletrotécnica e de Computadores

Candidato: José Afonso Laranjeira Pina, Nº 1070253, 1070253@isep.ipp.pt
Orientação científica: Lino Manuel Baptista Figueiredo, LBF@isep.ipp.pt
Coorientação científica: Ricardo Manuel Soares Anacleto, RIA@isep.ipp.pt

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
21 de outubro de 2015

Agradecimentos

À minha família pelo apoio incondicional e a todos os que de alguma forma me encorajaram na realização deste projeto.

Ao Engenheiro Lino Figueiredo e Engenheiro Ricardo Anacleto, meus orientadores, pelas suas críticas construtivas e sugestões, bem como pela dedicação e total disponibilidade.

Resumo

Atualmente os sistemas *Automatic Vehicle Location* (AVL) fazem parte do dia-a-dia de muitas empresas. Esta tecnologia tem evoluído significativamente ao longo da última década, tornando-se mais acessível e fácil de utilizar.

Este trabalho consiste no desenvolvimento de um sistema de localização de veículos para smartphone *Android*. Para tal, foram desenvolvidas duas aplicações: uma aplicação de localização para *smarphone Android* e uma aplicação WEB de monitorização. A aplicação de localização permite a recolha de dados de localização GPS e estabelecer uma rede *piconet Bluetooth*, admitindo assim a comunicação simultânea com a unidade de controlo de um veículo (ECU) através de um adaptador OBDII/*Bluetooth* e com até sete sensores/dispositivos *Bluetooth* que podem ser instalados no veículo. Os dados recolhidos pela aplicação *Android* são enviados periodicamente (intervalo de tempo definido pelo utilizador) para um servidor Web

No que diz respeito à aplicação WEB desenvolvida, esta permite a um gestor de frota efetuar a monitorização dos veículos em circulação/registados no sistema, podendo visualizar a posição geográfica dos mesmos num mapa interativo (*Google Maps*), dados do veículo (OBDII) e sensores/dispositivos *Bluetooth* para cada localização enviada pela aplicação *Android*. O sistema desenvolvido funciona tal como esperado. A aplicação *Android* foi testada inúmeras vezes e a diferentes velocidades do veículo, podendo inclusive funcionar em dois modos distintos: *data logger* e *data pusher*, consoante o estado da ligação à Internet do *smartphone*. Os sistemas de localização baseados em smartphone possuem vantagens relativamente aos sistemas convencionais, nomeadamente a portabilidade, facilidade de instalação e baixo custo.

Palavras-Chave

Android, Aplicação, AVL, *Bluetooth*, GPS, Localização, *Piconet*, Servidor, SQL, WEB.

Abstract

Currently fleet management systems or Automatic Vehicle Location Systems (AVL) are used by many companies on a daily basis. Over the last decade this technology has significantly evolved, making it more accessible and easy to use. Vehicle fleet management arose from companies needs to reduce costs and improve their services. The main purpose of this work is to develop a vehicle location system for Android smartphones. Two applications have been developed: an Android location application and a WEB monitoring application.

Android location application allows to collect GPS location data and establish a piconet Bluetooth network, thus permitting simultaneous communication with a vehicle Engine Control Unit through an OBDII/Bluetooth adapter, and up to seven Bluetooth sensors/devices that can be installed on the vehicle. Data collected by the Android application is sent periodically (user defined time interval) to a WEB server, where it is stored in a MySQL database for later viewing. The developed WEB application allows a vehicle fleet manager to monitor vehicles registered in the system. It is possible to verify a vehicle geographical position on an interactive map (Google Maps), vehicle (OBDII) and sensors/Bluetooth devices data for each location sent by the Android application.

The developed system works as expected. The Android application was tested multiple times at different vehicle speeds, and effectively it works in two different modes: data logger and data pusher, depending on the smartphone Internet connection status.

Location systems based smartphone, have advantages over conventional systems, including portability, ease of installation and low cost.

Keywords

Android, Application, AVL, Bluetooth, GPS, Location, Piconet, Server, SQL, WEB.

Índice

Índice de Figuras

Índice de Tabelas

Acrónimos

ADT – Android Development Tools

API – Application Programming Interface

AVL – Automatic Vehicle Location

ASCII – American Standard Code for Information Interchange

DOM – Document Object Model

DTC – Diagnostic Trouble Code

ECU – Engine Control Unit

HTTP – Hypertext Transfer Protocol

GPRS – General packet radio service

GPS – Global Positioning System

GSM – Global System for Mobile Communications

JSON – JavaScript Object Notation

OBD – On Board Diagnostic

PHP – HyperText Preprocessor

SDK – Software Development Kit

SQL – Structured Query Language

UI – User Interface

WEB – World Wide Web

1. INTRODUÇÃO

Uma frota de veículos automóveis é um conjunto de veículos que são normalmente utilizados para o transporte de passageiros ou mercadorias.

Os veículos de uma empresa de aluguer de automóvel, de táxis, de transporte de passageiros ou mercadorias, são exemplos de frotas automóveis.

Com os avanços da tecnologia, os sistemas de gestão de frota ou sistemas *Automatic Vehicle Location* (AVL) são comuns no dia-a-dia de muitas empresas. Esta tecnologia evoluiu significativamente ao longo da última década, tornando-se mais acessível e fácil de utilizar.

A concorrência crescente nos mercados globais é um desafio para as empresas, que necessitando de melhorar continuamente a qualidade dos serviços prestados, apostam neste tipo de sistemas tais são os benefícios, como por exemplo:

- Maior fiscalização no serviço de transporte – possibilidade de acompanhamento em tempo real das atividades realizadas;
- Redução do número de veículos – a otimização dos percursos efetuados;
- Correto uso do veículo – evitando o uso do veículo para fins pessoais ou outros que não sejam de interesse da empresa;
- Recuperação de veículos – Facilita a recuperação do veículo em caso de roubo;

- Administração da frota por dispositivo móvel. – Independentemente de onde o gestor esteja, os dados podem ser enviados para um dispositivo móvel, facilitando o trabalho do gestor;

1.1. MOTIVAÇÕES

Este projeto surgiu da intenção de realizar um trabalho no âmbito da programação de aplicações para dispositivos móveis (*smartphones*) com sistema operativo *Android,* com o intuito de adquirir/desenvolver competências nesta área de forte expansão.

O primeiro contacto com a tecnologia *Android* surgiu na disciplina de Laboratório de Mecatrónica, enquadrada no 2º ano do Mestrado em Engenharia Eletrotécnica e de Computadores, tendo sido desenvolvido um sistema de localização de veículos para *smartphone Android*. Um sistema de localização de veículos é geralmente associado a um dispositivo dedicado em *hardware*, algo dispendioso e que requer uma instalação árdua se for pretendido que o sistema adquira dados do veículo. Este é também um dos motivos que levou à escolha do tema deste trabalho, dado que atualmente este processo pode ser simplificado com o recurso a novas tecnologias e a um custo mais reduzido.

1.2. OBJETIVOS

Este projeto pretende reformular o sistema anteriormente desenvolvido, introduzindo novas capacidades ao sistema utilizando diferentes tecnologias.

É pretendido que o sistema admita, para além de permitir a um utilizador obter a posição geográfica de um dado veiculo, a possibilidade de aquisição de dados do motor do veículo e de um conjunto de sensores, para por exemplo, conhecer as condições atuais de condução do mesmo ou verificar a temperatura do compartimento de carga, respetivamente. A comunicação com o veículo e com os sensores será efetuada através das tecnologias *OBDII* e *Bluetooth.* Dada a complexidade inerente a este objetivo, sentiu-se a necessidade de o dividir em múltiplas tarefas, tais como:

- Pesquisa do estado da arte em sistemas AVL;

- Estudo genérico da plataforma *Android* – SDK;
- Estudo da tecnologia GPS;
- Estudo da tecnologia *Bluetooth;*
- Estudo do sistema OBDII e microcontrolador ELM327;
- Estudo do Google *Maps Javascript API;*
- Estudo de tecnologias WEB;
- O desenvolvimento de uma aplicação *Android* de localização para o veículo;
- O desenvolvimento de uma aplicação WEB de monitorização.

1.3. CALENDARIZAÇÃO

A Tabela 1 apresenta a calendarização prevista para o primeiro semestre.

Tabela 1 – Calendarização prevista para o primeiro semestre

	Setembro		**Outubro**					**Novembro**				**Dezembro**		
	1º	2ª	3ª	4ª	5ª	6ª	7ª	8ª	9ª	10ª	11ª	12ª	13ª	14ª
Pesquisa do estado da arte em sistemas AVL														
Estudo genérico do Android – SDK														
Estudo do Google Maps – API														
Desenvolvimento da aplicação Android (localização)														
Desenvolvimento da aplicação Android (Monitorização)														
Fase de teste														
Melhoramentos														

A Tabela 2 apresenta a calendarização prevista para o segundo semestre.

Tabela 2 – Calendarização prevista para o segundo semestre

Descrição	Jan.		Fev.		Mar.		Abril		Maio		Junho		Julho		Agosto		Set.	
Estudo da tecnologia *Bluetooth*																		
Estudo do sistema OBDII e microcontrolador ELM327																		
Estudo do sistema operativo *Android e* API																		
Desenvolvimento da aplicação de localização *Android e sensores*																		
Estudo do *Google Maps Javascript API*																		
Estudo de tecnologias WEB																		
Desenvolvimento da aplicação WEB																		
Fase de teste																		

1.4. ORGANIZAÇÃO DO RELATÓRIO

No Capítulo 1 é descrito o tema do trabalho a desenvolver, objetivos e motivações. No capítulo seguinte, 2, são apresentados alguns sistemas de localização de veículos existentes quanto à sua arquitetura, modos de funcionamento e principais características. No Capítulo 3 são descritas as principais tecnologias que serão utilizadas no desenvolvimento do sistema. No capítulo seguinte, 4, é descrita a implementação do sistema. No capítulo 5 é feita a análise ao sistema desenvolvido. No último capítulo, o 6º, são reunidas as principais conclusões e perspetivados futuros desenvolvimentos.

2. SISTEMAS DE LOCALIZAÇÃO DE VEICULOS

Em empresas de grande dimensão com um elevado número de viaturas, a complexidade da gestão destas torna-se evidente. Tendo em conta este objetivo foram desenvolvidas soluções, mais concretamente os sistemas AVL (*Automatic Vehicle Locator*), de forma a permitir uma gestão mais facilitada. A Figura 1 ilustra uma possível taxonomia de um sistema AVL. Este tipo de sistemas que podem ser seccionados em duas áreas principais: os sistemas instalados nos veículos e os sistemas de monitorização (central de controlo ou servidor de dados).

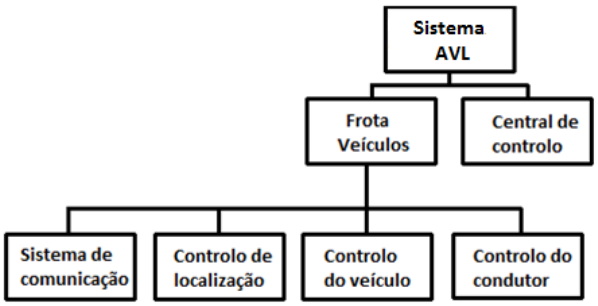


Figura 1 – Taxonomia de um sistema AVL

As soluções existentes no mercado podem permitir apenas a localização de um veículo, o controlo de sistemas integrados (ex.: controlo da temperatura da carga ou imobilização do veiculo em caso de furto) ou até mesmo o controlo do condutor (análise do estilo de condução, por exemplo). Existe um número bastante elevado de funcionalidades que podem ser integradas neste tipo de sistemas.

2.1. MODOS DE FUNCIONAMENTO

Um sistema AVL pode genericamente funcionar em um de três diferentes modos [1]: *Data pusher*, *Data puller* e *Data logger*. A Figura 2 ilustra a arquitetura de um sistema AVL nos modos de funcionamento *Data puller e Data pusher*.

Figura 2 – Arquitetura de um sistema AVL (em modo *Data puller* ou *Data pusher*) [2]

2.1.1. *DATA PUSHER*

Este é o modo de funcionamento mais comum. O veículo a monitorizar possui um dispositivo que incorpora um módulo GPS e um *modem* GSM.

Após a determinação da posição geográfica do veículo, este parâmetro (podem ser recolhidas outras informações como velocidade ou altitude) é enviado através do *modem* GSM em intervalos regulares, via SMS ou GPRS para um servidor que armazena os dados enviados.

2.1.2. *DATA PULLER*

Contrariamente ao modo data *pusher* que envia a posição do veículo em intervalos de tempo regulares, este modo de funcionamento permite que a localização do dispositivo seja consultada apenas quando necessário, enviando geralmente como resposta, uma mensagem SMS a um pedido de localização.

2.1.3. *DATA LOGGER*

Um sistema neste modo de funcionamento regista a posição do dispositivo em intervalos regulares e grava esta informação no próprio dispositivo, em memória interna ou externa. Posteriormente o utilizador pode visualizar e/ou analisar estes dados num computador.

Um sistema que apenas funcione neste modo, não pode ser considerado um sistema AVL, no entanto, alguns sistemas podem funcionar neste modo em caso de perda de sinal GSM, efetuando a sincronização de dados logo que a ligação esteja restabelecida.

2.2. CONTROLO DE LOCALIZAÇÃO

O *Global Positioning System* (GPS) é o mais conhecido e utilizado sistema de localização em sistemas AVL.

GPS é a abreviatura de NAVSTAR GPS (*NAVSTAR GPS-NAVigation System with Time And Ranging Global Positioning System*). É um sistema de radionavegação baseado em satélites, desenvolvido e controlado pelo departamento de defesa dos Estados Unidos da América.

Este sistema possibilita ao utilizador de um dispositivo GPS recetor determinar a sua localização geográfica, assim como obter informação horária, em qualquer ponto da Terra, desde que o recetor se encontre no campo de visão de pelo menos quatro satélites GPS. Atualmente o sistema GPS é constituído por três principais componentes: espacial, controlo e de utilizador [3]. A componente espacial é constituída por uma constelação de 24 satélites em órbita terrestre, a uma altitude de aproximadamente 20200 km, com um período de 12h siderais e distribuídos por seis planos orbitais com quatro satélites cada [3].

Estes planos estão separados entre si por cerca de 60º em longitude e têm inclinações de cerca de 55º em relação ao plano equatorial terrestre, Figura 3.

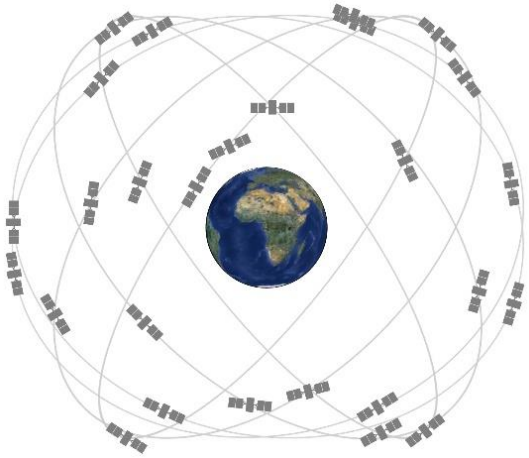


Figura 3 – Constelação de satélites GPS [3]

O sistema GPS foi desenhado de forma a possibilitar a visibilidade de pelo menos quatro satélites, acima da linha do horizonte, em qualquer ponto da superfície terrestre e a qualquer momento.

A componente de controlo é constituída por cinco estações de monitorização distribuídas ao longo do Terra e de uma estação de controlo principal. Esta componente monitoriza e analisa os sinais emitidos pelos satélites em órbita comparando os mesmos com modelos padrão para validar a posição orbital de cada satélite.

A componente do utilizador inclui todos aqueles que usam um recetor GPS, podendo ser divididos em dois tipos, militar e civil. O Exército dos EUA utiliza GPS para sistemas de navegação, reconhecimento e orientação de mísseis. A maioria dos usos civis de GPS, no entanto, enquadra-se numa de quatro categorias: navegação, topografia, cartografia e cronometragem.

A posição de um recetor GPS é calculada através de uma técnica denominada *satellite ranging*, que implica a medição da distância entre o recetor GPS e os satélites GPS que está a monitorizar [4],[5].

A distância calculada a um só satélite permite limitar a posição do dispositivo recetor ao raio de uma esfera. A distância a um segundo satélite permite reduzir a incerteza à intersecção de duas esferas.

Calculando a distância a um terceiro satélite a interseção das três esferas resume-se a dois pontos, Figura 4.

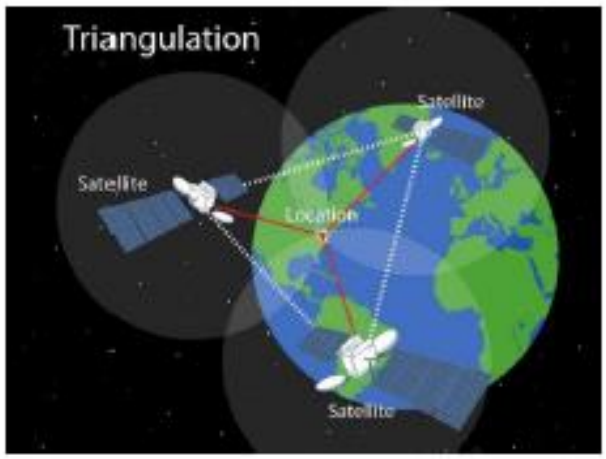


Figura 4 – Técnica *satellite ranging [6]*

Geralmente um destes pontos encontra-se muito afastado da Terra (ou com uma velocidade muito elevada), o que permite que a posição do dispositivo seja determinada por exclusão de partes [4]. O quarto satélite é utilizado como auxiliar, enviando ao dispositivo recetor um sinal que o ajuda a determinar o instante de tempo em que ocorrem as emissões, o que evita a utilização de um relógio atómico na determinação do tempo.

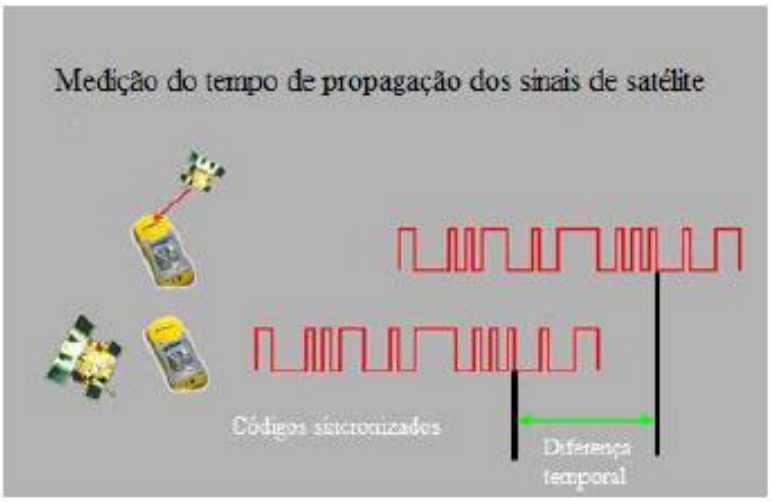


Figura 5 – Sicronização dos códigos satelite/receptor [5]

A medição de distância entre o recetor GPS e um satélite é efetuada medindo o tempo necessário que o sinal emitido pelo satélite necessita para alcançar o recetor. Sabendo esse tempo de propagação e que o sinal se desloca à velocidade da luz, é possível o cálculo da

distância. Cada satélite possui uma sequência própria que é enviada ao recetor. O satélite e o recetor geram pseudo-códigos no mesmo instante. Efetuando a sincronização destes códigos, Figura 5, o atraso na receção pode ser calculado. Esse atraso multiplicado pela velocidade de propagação é igual à distância à qual se encontra o recetor do satélite.

2.3. SISTEMA DE COMUNICAÇÃO

Um sistema AVL funciona com base na comunicação entre veículos e um centro de controlo ou servidor de receção de dados. Com os avanços das comunicações móveis nos últimos anos, o sistema GSM tornou-se bastante difundido tornando-se assim, no principal meio de comunicação utilizado em sistemas de gestão de frota.

O sistema GSM (*Global System for Mobile Communications*) é um sistema de dados/voz celular digital de alta qualidade e é o *standard* global para comunicações móveis, com uma quota de mercado superior a 90% e disponível em mais de 219 países e territórios [7].

O GSM pode ser utilizado para efetuar chamadas de voz, envio e receção de dados, SMS (*Short Message Service*), entre outros. Para utilizar o serviço GSM é necessário existir cobertura de sinal de uma operadora de serviços móveis GSM. Geralmente nos sistemas AVL, apenas são utilizados os serviços GSM de dados e de SMS. A Figura 6 ilustra um exemplo de um recetor GSM que pode ser instalado num veículo.

Figura 6 – Receptor GSM

A comunicação com estes módulos faz essencialmente através de comandos *ATtention* (AT). O "AT" é um prefixo que indica ao *modem* o início de uma instrução, tal como "ATD" (Marcar) ou "ATA" (Responder). Para além dos comandos AT normais, os *modems* GSM suportam um outro tipo de comandos "AT" específicos da tecnologia GSM, que incluem comandos como: "AT+CMSS" (Enviar SMS), "AT+CMGR" (Ler SMS) ou "AT+CMGL"

(listar mensagens SMS). Os comandos que se iniciam com o caracter "+" são denominados de *extended commands*.

2.4. TIPOS DE SISTEMA

Atualmente estão disponíveis diversos tipos de sistemas AVL. Estes podem ser divididos em dois tipos:

- Sistemas dedicados;
- Sistemas baseados em dispositivos móveis.

2.4.1. SISTEMAS DEDICADOS

Um típico sistema dedicado de localização é composto por um dispositivo de localização por GPS que é instalado no veículo, e um servidor/Interface do utilizador.

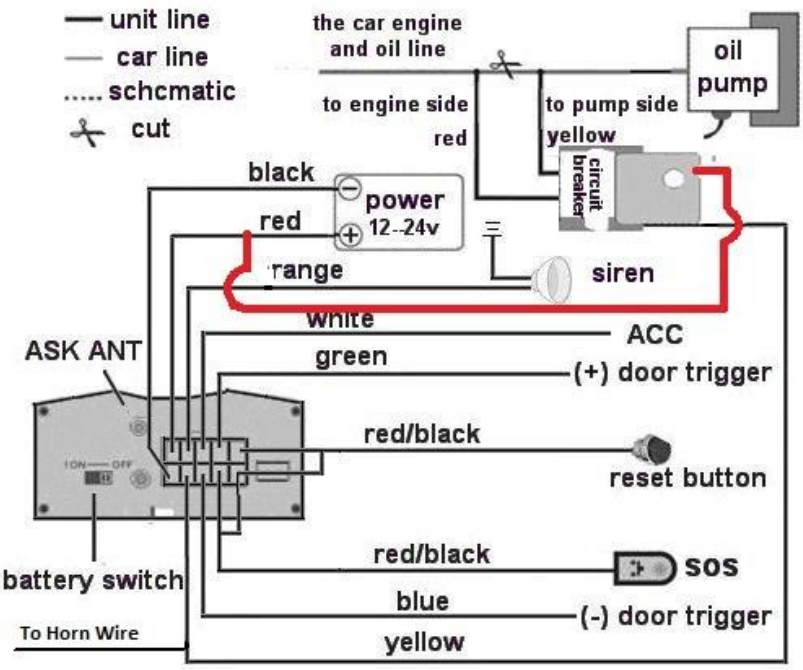


Figura 7 – Esquema de ligações típico de um dispositivo de localização de veículos [8]

Este dispositivo normalmente fornece mais informações acerca do veículo do que apenas a sua localização, podendo analisar a quantidade de combustível, estado do botão de

emergência, temperatura do motor, altitude, indicação de porta aberta, estado da bateria, a rotação do motor, posição do acelerador, número de satélites GPS em vista, entre outros.

A Figura 7 ilustra o esquema de ligações de um dispositivo de localização (*TK103 GPS Tracker*). Pode ser constatado que dependendo do número de sensores e/ou atuadores, que dependem fundamentalmente do número de funcionalidades do sistema, a instalação pode tornar-se bastante complexa. Alguns destes dispositivos permitem também o controlo remoto de algumas funções do veículo, tais como o corte de combustível, fecho e abertura de portas, ligar o veículo, acionamento do alarme, entre outros.

De modo a simplificar a instalação, alguns fabricantes, como o caso da *TomTom*, fornecem dispositivos com conexão CAN-BUS (para ligação a um interface FMS, que permite a um sistema AVL o acesso a informações de um veículo compatível, independentemente do modelo ou fabricante), como ilustra a Figura 8. Isto diminui dramaticamente o tempo e complexidade da instalação.

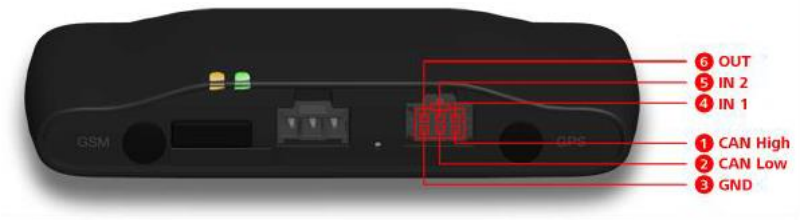


Figura 8 – TomTom Link 510 – Esquema de ligações [10]

A Figura 9 ilustra um exemplo da arquitetura de *hardware* deste tipo de dispositivos, no caso, um produto da empresa *Tackjinn*. Este dispositivo é composto por 4 módulos principais: um módulo GPS, módulo GSM, módulo de alimentação e um processador [9]. Outros módulos podem ser utilizados dependendo do modelo do equipamento e sua finalidade.

- GSM/GPRS Modem - Efetua a comunicação de dados com o servidor
- GPS Module: - Recebe o sinal de GPS
- Processador – Efetua o controlo e processamento do Sistema (GPS, sensores, GSM, I/O e GPRS)

- Memória *flash* – Utilizada para armazenar os dados provenientes do módulo GPS e sensores caso não haja uma ligação GPRS ativa.

- Módulo de alimentação – Fornece a alimentação aos diferentes componentes do sistema

- Sensores – Monitorização de parâmetros tais como a quantidade de combustível, estado do botão de emergência, temperatura do motor, entre outros

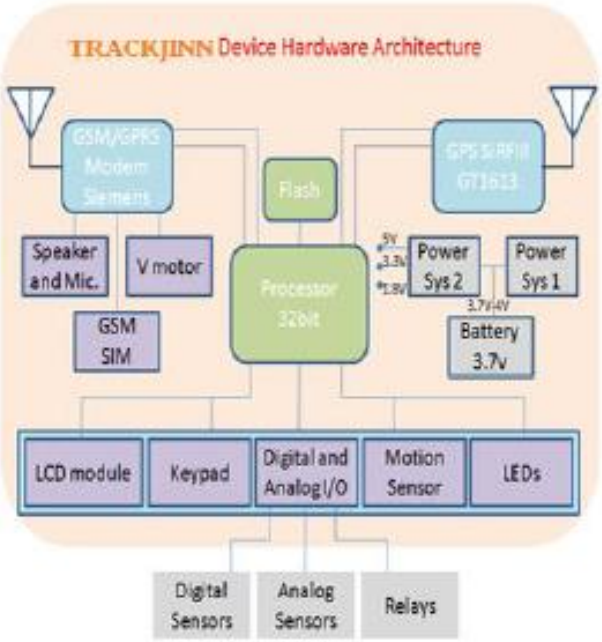


Figura 9 – Arquitetura de *hardware* – Trackjinn [9]

No que diz respeito ao servidor, este recebe os dados provenientes da unidade de localização por GPS e disponibiliza-os posteriormente ao utilizador normalmente através de uma aplicação WEB.

Todos os dados enviados pelo veículo estão disponível para consulta, sendo possível a elaboração de gráficos da velocidade do veículo, consumo de combustível, entre outros.
As empresas fornecedoras destes sistemas oferecem tipicamente uma solução completa para o sistema (dispositivo para o veiculo e software de gestão).

No caso de o utilizador pretender um outro tipo de serviço de gestão, existem soluções no mercado, como por exemplo o serviço da empresa *gSat Smart Connections,* que

disponibiliza um servidor e respetivo interface para o utilizador, compatível com um enorme número de dispositivos de diferentes marcas [11], Figura 10.

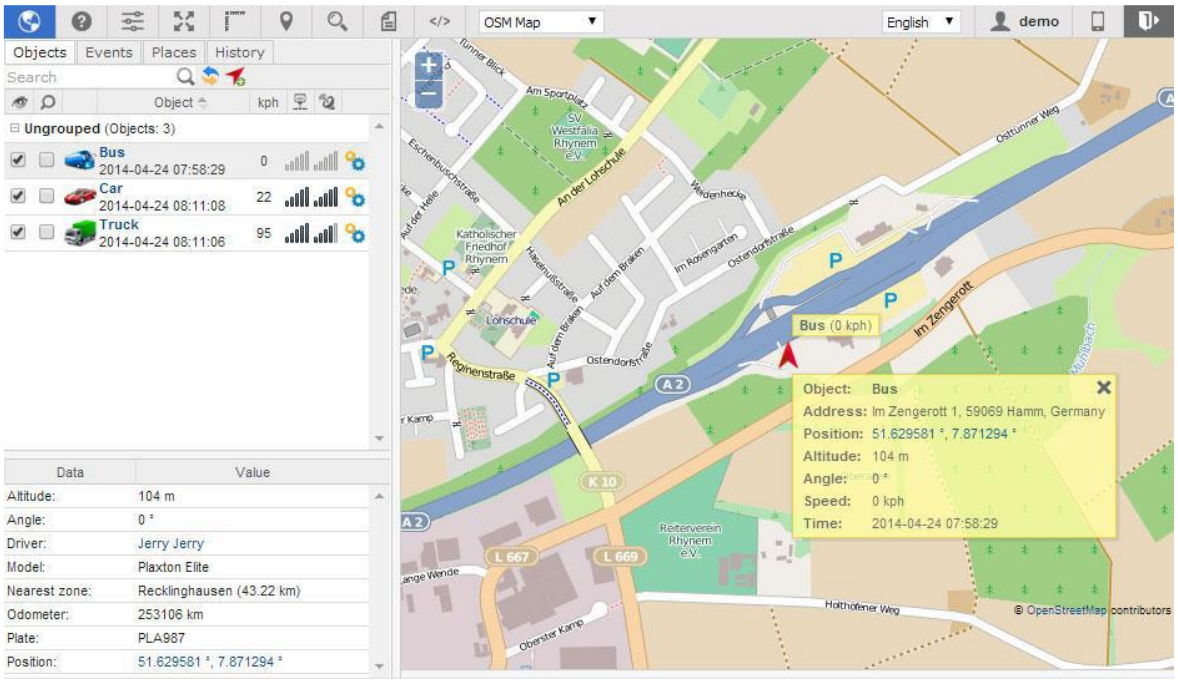


Figura 10 – Aplicação Web de monitorização *gSat Smart Connections [11]*

2.4.2. SISTEMAS BASEADOS EM DISPOSITIVO MÓVEL

Atualmente é notável a presença e o impacto dos dispositivos móveis (como *tablets*, *smartphones* e *PDA's*) na vida contemporânea. Temos como exemplo os *smartphones*, que podem ser definidos como telemóveis que oferecem recursos avançados, como é o caso do GPS integrado.

Dada as capacidades e funcionalidades destes dispositivos, e de muitos destes equipamentos estarem equipados com módulos GPS, surgiram no mercado aplicações para *smartphones* que permitem o seu uso como sistemas de navegação e/ou localização, Figura 12.

A arquitetura destes sistemas é semelhante à dos sistemas dedicados, no entanto, não necessita de instalação no veículo, o que permite a utilização do sistema com múltiplos veículos com o mesmo dispositivo móvel (dependendo das funcionalidades da aplicação). Os *smartphones* com GPS integrado ou externo podem funcionar como um dispositivo do tipo *data logger*, *data pusher* ou *data puller*, executando aplicações específicas,

independentemente da plataforma, *iPhone*, *Android*, *Windows Mobile* e *Symbian*. Para o sistema *Android* estão disponíveis várias aplicações para a localização de veículos, no entanto o seu funcionamento é similar, com ligeiras diferenças ao nível das funcionalidades oferecidas aos utilizadores. A aplicação *mycartracks* para dispositivos móveis *Android*, Figura 11, é um exemplo de um sistema AVL:

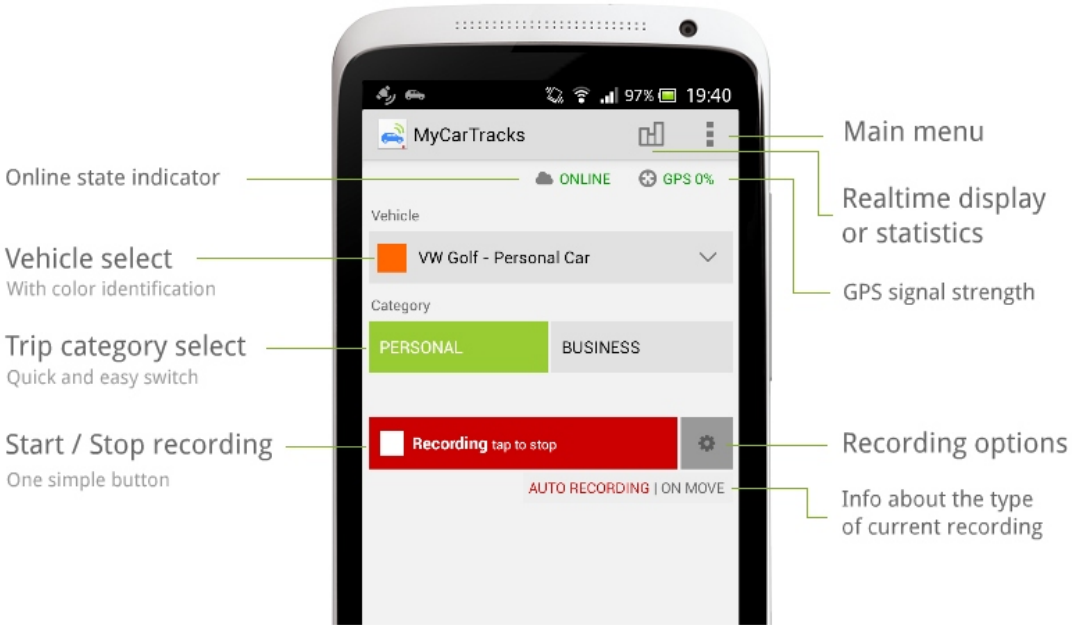


Figura 11 – Aplicação Android *mycartrack[12]*

Algumas das funcionalidades da aplicação *mycartrack [12]*:

- Menu de seleção de veículos – Permitindo o uso da aplicação com diferentes veículos;
- Opções de gravação – Gravação manual ou automática;
- *Start/Stop* – Iniciar e interromper a gravação;
- Tipo de viagem – Permite distinguir o uso pessoal do uso profissional da viatura;
- *Offline tracking* – Caso o utilizador não disponha de um plano de dados para comunicação com o servidor, a aplicação permite a sincronização posterior de dados através de uma ligação Wi-fi de modo a diminuir os custos para o utilizador.

Tal como nos sistemas dedicados as empresas que desenvolvem estas aplicações oferecem uma solução completa para o sistema, disponibilizando aos clientes uma aplicação WEB de gestão dos dados obtidos, Figura 12.

Este tipo de sistemas possuem no entanto, algumas possíveis desvantagens em relação aos sistemas tradicionais (dedicados):

- Tendo este tipo de sistemas como base um dispositivo móvel, um supervisor da frota tem de agir com base na confiança, já que é possível desligar a aplicação ou mesmo o dispositivo;
- O investimento num sistema dedicado é superior ao de um smartphone. No entanto a sua durabilidade é bastante superior, e raramente necessitam de se substituídos, ao contrário de um dispositivo móvel deste tipo.

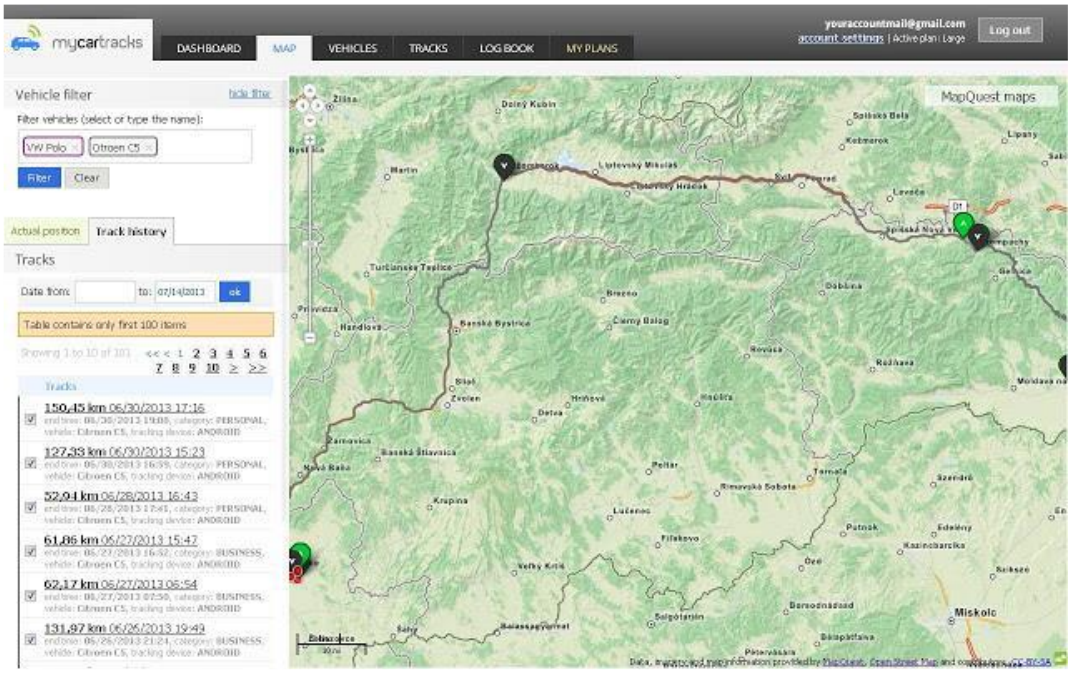


Figura 12 – Aplicação WEB de gestão [12]

2.5. *FLEET MANAGEMENT SYSTEMS INTERFACE*

O *Fleet Management Systems Interface* (FMS) é uma interface ou dispositivo, Figura 13, que permite a comunicação entre veículos e sistemas de localização. O FMS-Standard foi desenvolvido, em 2002, por seis grandes fabricantes europeus como *Daimler* AG, MAN AG, Scania, DAF *Trucks* e IVECO com o objetivo de criar um interface que permitisse a um sistema de localização aceder a informações sobre um veículo independentemente do modelo ou fabricante [13], como ilustra a Figura 14.

Esta interface permite o acesso aos seguintes dados (FMS standard):

- A posição do acelerador, travão e embraiagem;
- Total de combustível utilizado (desde início de ciclo de vida do veículo);
- Nível de combustível no depósito (0-100%);
- Velocidade rotação do motor e velocidade do veículo
- Peso do eixo (em kg);
- Número ou código de identificação do veículo;
- Distância percorrida pelo veículo.

Figura 13 – FMS Gateway [13]

Algumas funcionalidades adicionais:

- Código de identificação do condutor;
- Tempo de condução;
- Informação de diagnóstico do veículo.

No caso da interface visível na Figura 13, esta liga ao barramento CAN do veículo através de um adaptador *contactless*, denominado CL-CAN, como ilustrado na Figura 14. Posteriormente o *interface* fornece acesso a um sistema AVL através do protocolo CAN, RS232 ou ligação *Bluetooth*, segundo o protocolo FMS *Standard*, definido no SAE J1939.

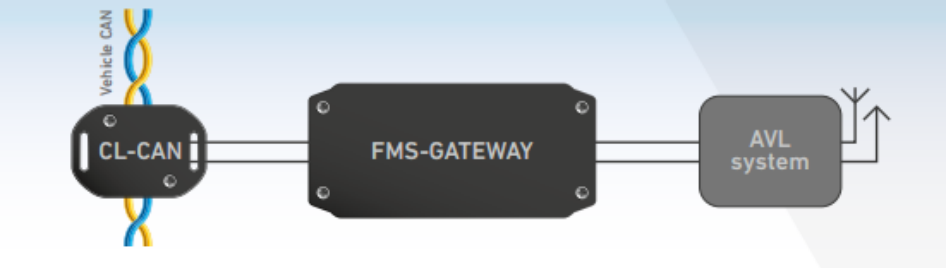


Figura 14 – FMS Gateway CAN-AVL [13]

3. ANÁLISE DAS TECNOLOGIAS ENVOLVIDAS

Este capítulo pretende analisar algumas das tecnologias particularmente importantes para o desenvolvimento deste trabalho.

3.1. *BLUETOOTH*

O *Bluetooth* é uma tecnologia de comunicação sem fios para curtas distâncias, destinada a substituir os cabos que interligam dispositivos, mantendo elevados níveis de segurança e tendo como principais características a robustez, baixo consumo energético e o baixo custo.

O *Bluetooth Special Interest Group* (SIG) é o consórcio responsável pelo protocolo. Este surgiu em 1998 formado pelas empresas Ericsson, Intel, IBM, Toshiba e Nokia [14]. Esta tecnologia é utilizada em bilhões de dispositivos tais como telefones, computadores ou produtos de entretenimento doméstico. Uma das principais vantagens desta tecnologia é a possibilidade de lidar simultaneamente com transmissões de dados e voz, fornecendo aos seus utilizadores variadas soluções.

A tecnologia *Bluetooth* oferece a possibilidade de ligações tanto ponto-a-ponto como ponto-a-multiponto. Quando dois ou mais dispositivos estão ligados entre si, obtém-se uma rede *ad-hoc*, conhecida por rede *piconet.*

Uma rede *piconet* consiste em oito dispositivos *Bluetooth* interligados (uma unidade mestre e até sete escravos). O dispositivo mestre é a unidade que inicia as transmissões e os escravos são unidades de resposta. Este tipo de rede *Bluetooth* pode ter apenas uma unidade mestre. Qualquer unidade numa rede *piconet* pode comunicar com uma segunda *piconet*, no entanto pode apenas funcionar como mestre numa *piconet* de cada vez. Se várias redes *piconet* se sobrepuserem na mesma área, Figura 15, e os membros das várias *piconets* poderem comunicar entre si, esta nova rede, é conhecida como uma *scatternet*.

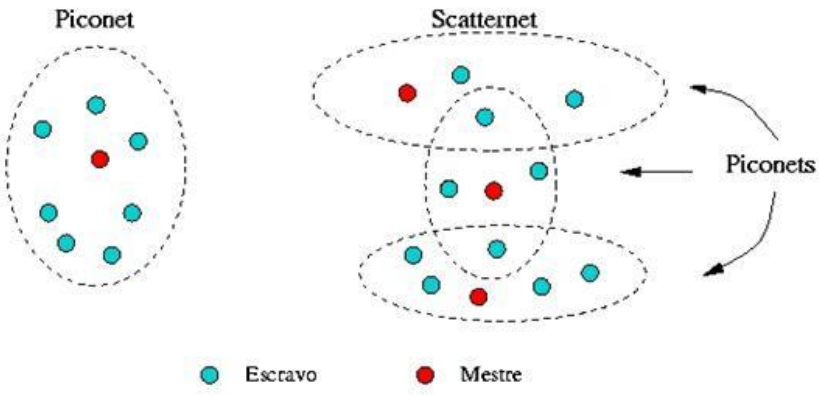


Figura 15 – Redes *Piconet* e *Scatternet [15]*

Quando à potência os dispositivos *Bluetooth* dividem-se em 3 classes, que especificam a potência de saída da antena, Tabela 3. Os dispositivos de classe 2 são os mais comuns e os que oferecem um melhor rácio potência – alcance.

Tabela 3 – Classe dos dispositivos *Bluetooth*

Classe	Potência Máxima	Alcance Máximo
1	100mW	100m
2	2.5mW	10m
3	1mW	1m

3.1.1. ARQUITETURA

A arquitetura *Bluetooth* consiste em dois principais componentes: o *hardware* e uma pilha de protocolos (*software*) [16]. A arquitetura *Bluetooth* e as suas características técnicas estão definidas em duas especificações: *Core* (Núcleo) e *Profiles* (Perfis). A especificação do núcleo define o funcionamento do sistema (como protocolos, camadas, e especificações técnicas), enquanto a especificação dos perfis define e fornece protocolos que os fabricantes devem seguir, de modo a assegurar a interoperabilidade entre dispositivos *Bluetooth* de diferentes fabricantes.

A pilha de protocolos na especificação *Bluetooth* é divida em duas principais partes: inferior e superior [17]. A parte inferior é composta pela camada de radio, a camada *Baseband* e *Link Manager*, enquanto na parte superior estão contidas a camada *Logical Link Control and Adaptation Layer (*L2CAP), *Radio Frequency Communication* (RFCOMM), entre outros.

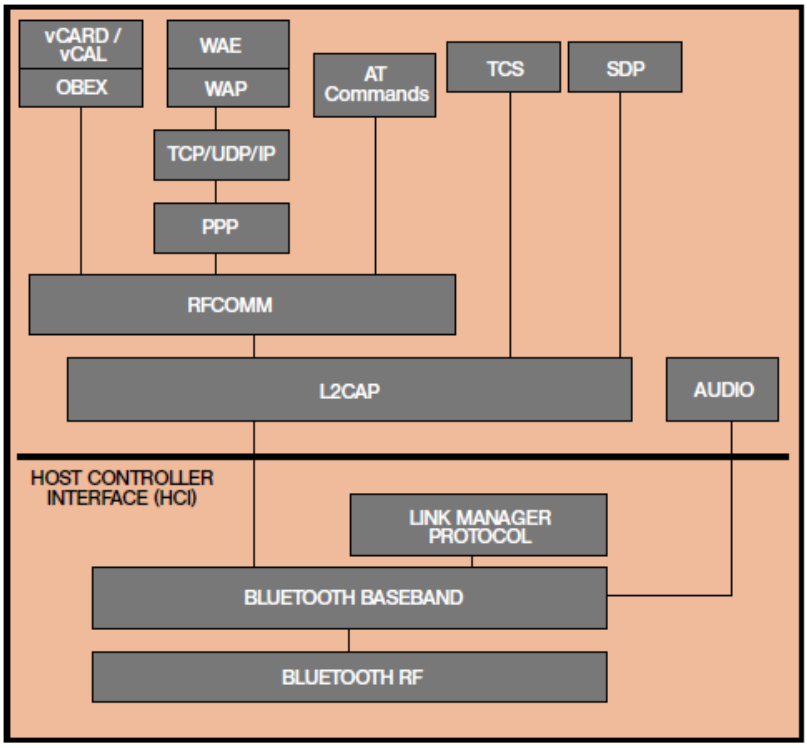


Figura 16 – *Bluetooth* – Pilha protocolar [17]

1. **Rádio (*Bluetooth Radio*)**

A camada *Radio* define a frequência, potência e modulação utilizada pela antena *Bluetooth*. A tecnologia *Bluetooth* atua na faixa não-licenciada industrial, científica e médica (ISM), na gama de frequências 2.4 GHz – 2,485 GHz e sinal *full-duplex*. A taxa de transmissão pode

atingir 3Mbps com um mecanismo recentemente introduzido na última especificação, *Enhanced Data Rate*. De forma a diminuir interferências por parte de dispositivos na mesma gama de frequências, é utilizada a tecnologia AFH (*Adaptative Frequency Hopping*). Esta tecnologia permite detetar outros dispositivos e alternar de canais de transmissão a uma taxa 1600 saltos por segundo (79 canais disponíveis, espaçados de 1Mhz), evitando assim frequências já utilizadas.

2. BaseBand

A camada *Baseband* define como os dispositivos *Bluetooth* se localizam e interligam, para além de outros serviços, como o tipo de pacotes, processamento de pacotes, deteção de erros, criptografia, transmissão e retransmissão de pacotes.

Nesta camada são definidas as funções do *master* e do *slave* bem como o padrão de saltos de frequência utilizados pelos dispositivos. Os *slaves* partilham o relógio do *master* e a transmissão é feita através de *slots* de tempo pré-definidos. Os *slots* são canais divididos em períodos de 625 microssegundos sendo que a cada salto de frequência corresponde um *slot*. Esta camada suporta dois tipos de *links*: *Synchronous Connection-Oriented* (SCO) e *Assynchronous Connection-Less* (ACL) [15].

Os *links* SCO caracterizam-se pela atribuição de um *slot* de tempo a um dispositivo periodicamente. Este tipo de *links* são utilizados essencialmente para transmissão de voz, o que requer transmissões rápidas e consistentes de dados. Um dispositivo que estabeleceu um *link* SCO possui *slots* de tempo reservados para o seu uso. Estes pacotes são prioritários e são processados antes de pacotes ACL. Dado o seu modo de funcionamento, o SCO não permite a retransmissão de pacotes (dados perdidos) [15]. Um dispositivo que opera sobre um *link* ACL pode enviar pacotes de tamanho variável de 1, 3 ou 5 *slots* de tempo. Entretanto, este tipo de *link* não possui reserva de *slots* de tempo para os seus pacotes [16]. Ao contrário de um *link* SCO, o ACL permite o reenvio de pacotes (dados perdidos) [15];

3. Link Manager

Esta bloco implementa o *Link Manager Protocol* (LMP), que negocia todos os aspetos do funcionamento da conexão *Bluetooth* entre dois dispositivos, para alem de efetuar a gestão

da taxa de transferência de dados, taxa de transferência de áudio, autenticação, níveis de confiança entre dispositivos, criptografia de dados e adaptação da potência de transmissão.

4. Logical Link Control and Adaptation Protocol

O protocolo *Logical Link Control and Adaptation Protocol (*L2CAP) é o interface entre os protocolos das camadas superiores e os protocolos de transporte das camadas inferiores. Fornece multiplexagem de modo a que múltiplas aplicações possam partilhar as ligações com as camadas mais baixas. Esta camada é também responsável pela fragmentação de pacotes de dados para a posterior transmissão pela banda base e remontagem de pacotes destes nos dispositivos recetores.

5. *Host Controller Interface*

O *Host Controller Interface* (HCI) permite que as mais altas camadas da pilha protocolar incluindo aplicações, tenham acesso a camada *Baseband, Link manager* e outros registros de *hardware* através de uma interface padrão simples como por exemplo USB, RS-232 ou UART, proporcionando integração aos mais diversos componentes aos módulos *Bluetooth* de diferentes fabricantes.

6. Camadas superiores

A camada superior (acima da camada L2CAP) inclui protocolos de terceiros e padrões industriais que permitem que novas aplicações ou já existentes operem sobre ligações *Bluetooth* [16].

Os protocolos desta camada incluem por exemplo: *Point-to Point Protocol* (PPP), *Internet Protocol* (IP), *Transmission Control Protocol* (TCP), *Wireless Application Protocol* (WAP), entre outros. Outros protocolos desenvolvidos pelo SIG também foram incluídos tais como o *Service Discover Protocol* (SDP) e o RFComm.

O RFComm é um protocolo que permite emular as portas série e USB através do protocolo L2CAP. Foi projetado para permitir que dispositivos com uma tecnologia mais antiga o possam utilizar com facilidade.

3.1.2. ENDEREÇOS E NOMES *BLUETOOTH*

Cada dispositivo *Bluetooth* tem um endereço único de 48 bits associado, comumente abreviado BD_ADDR [18]. Normalmente este endereço será apresentado sob a forma de um valor hexadecimal com 12 dígitos. A metade mais significativa (24 bits) do endereço é o *Organization Unique Identifier* (OUI), que identifica o fabricante. Os restantes 24-bits são a parte original do endereço. Este endereço é visível na maior parte dos dispositivos como por exemplo: "MAC No.: 00:06:66:42:21:52".

3.1.3. PROCESSO DE LIGAÇÃO *BLUETOOTH*

A criação de uma ligação *Bluetooth* entre dois dispositivos é um processo que envolve três etapas [16]:

- ***Inquiry*** – Se dois dispositivos nunca estiveram interligados, é necessário executar uma busca num dos dispositivos na tentativa de detetar a presença do outro dispositivo. Neste caso um dos dispositivos envia um pedido de inquérito, e qualquer dispositivo que recebam o pedido irão responder com o seu endereço;
- ***Paging*** - É o processo de formação de uma ligação entre dois dispositivos *Bluetooth*. Antes de esta ligação poder ser iniciada, cada dispositivo precisa saber o endereço do outro (encontrado no processo de *Inquiry*).
- ***Connection*** – Após um dispositivo ter concluído o processo de *Paging*, este entra no estado de conexão. Enquanto conectado, um dispositivo pode participar ativamente ou pode ser colocado num modo de baixo consumo.

Num estado de ligação, um dispositivo *Bluetooth* tem diferentes modos de operação [16]:

- Modo Ativo – Neste modo o dispositivo está a transmitir ou a receber dados ativamente;
- Modo *Sniff* - Modo de baixo consumo utilizado por um dispositivo *slave*, por ordem do *master*. Neste modo o dispositivo apenas escuta a rede em intervalos de tempo fixos;

- Modo *Hold* – Neste modo o dispositivo *slave* fica inativo durante um certo período de tempo, sem qualquer transmissão de dados. Após esse período o aparelho reinicia as transmissões;
- Modo *Park* - Modo de menor consumo de energia. Neste modo o dispositivo adquire um endereço relativo ao modo *Park*. Este dispositivo ainda pertence à rede mas não participa na transmissão de dados.

3.1.4. ESPECIFICAÇÕES BLUETOOTH

Regularmente são feitas atualizações às versões *Bluetooth* existentes que adicionam novas funcionalidades, mantendo a compatibilidade com as versões anteriores. A troca de informações é possível entre diferentes versões, mas a taxa de transmissão está limitada pelo dipositivo com a versão mais baixa. Diferenças entre as principais versões *Bluetooth* [19]:

- *Bluetooth* v1.2 – As versões v1.x lançaram as bases para os protocolos e especificações de futuras versões. Os dispositivos com esta versão suportam taxas de transferência de até 1 Mbps e têm um raio de ação de cerca de 10 metros.
- *Bluetooth* v2.1 + EDR - Nas versões 2.x foi introduzida a tecnologia *Enhanced Data Rate* (EDR), o que aumentou a taxa de transferência de dados possível para cerca de 3 Mbps. A versão *Bluetooth* v2.1, lançada em 2007, introduziu o emparelhamento seguro simples (SSP).
- *Bluetooth* v3.0 + HS – Nesta versão a taxa de transferência de dados possível aumentou para 24 Mbps. No entanto os dados são na realidade transmitidos através de uma ligação Wi-Fi (802.11), sendo o *Bluetooth* utlizado apenas para estabelecer e gerir a ligação. Esta velocidade apenas é suportada nas versões 3.0 + HS, dado que os dispositivos não-HS estão limitados a velocidades máximas de transferência de 3Mbps.
- *Bluetooth* v4.0 – Esta versão *Bluetooth* divide a especificação *Bluetooth* em três categorias: *classic*, *high-speed* e *low-energy*. A categoria *classic* e *high-speed* utilizam as versões *Bluetooth* v2.1 + EDR e v3.0 + HS, respetivamente. A principal inovação é a categoria *low-energy* (BLE) para aplicações de baixa potência. De modo a reduzir o consumo dos dispositivos, tanto o alcance máximo como a taxa de transferência são reduzidas (50m e 0,27Mbps, respetivamente). Esta versão é

destinada a dispositivos que operam com baterias e que não requerem uma constante transmissão de dados/ou grandes taxas de transferência.

3.1.5. PERFIS DOS DISPOSITIVOS *BLUETOOTH*

A norma *Bluetooth* define um certo número de perfis de aplicativos, de forma a definir que tipo de serviços são oferecidos por um determinado dispositivo *Bluetooth*. Um dispositivo *Bluetooth* pode suportar vários perfis, entre os quais [16]:

- *Service Port Profile* (SPP) – Define como configurar as portas série para dois dispositivos usando o protocolo RFCOMM.
- *Human Interface Device Profile* (HID) – Define quais os protocolos, procedimentos e especificações para utilizar com teclados sem fios e outros dispositivos.
- *Video Distribution Profile* (VDP) – Define como os dispositivos efetuam *stream* de vídeo através de *Bluetooth*.
- *Generic Object Profile* (GOEP) – Utilizado para transferir objetos entre dispositivos.

3.2. *ON BOARD DIAGNOSTIC (OBD)*

O *On Board Diagnostic* (OBD) é um sistema eletrónico incorporado em veículos de passageiros, comerciais e pesados com motores de combustão interna [21]. Estes motores produzem emissões de gases tóxicos para a atmosfera, tais como o monóxido de carbono (CO), hidrocarbonetos, óxidos de nitrogênio (NOx) ou dióxido de enxofre.

Durante os anos 70 e início dos anos 80, os fabricantes de veículos começaram a utilizar meios eletrónicos para controlar as funções do motor e diagnosticar problemas no motor, principalmente para atender às normas de emissões de gases poluentes. Ao longo dos anos, os sistemas *On Board Diagnostic* tornaram-se mais sofisticados, oferecendo um controlo quase total dos parâmetros de funcionamento do motor e permitindo também monitorar outros componentes do veículo. Após novas regulamentações e exigências normativas quanto as emissões, a partir de 1996 entrou em vigor o sistema OBDII, um conjunto de normas e práticas desenvolvidas pelo SAE e pela *International Organization for Standardization* (ISO), que na atualidade vigora mundialmente.

O sistema OBD II foi desenvolvido com o objetivo de detetar que o desempenho de um determinado componente do sistema/conjunto propulsor se deteriorou a tal ponto que o veículo vai exceder o limite de emissões gases poluentes definidos por lei. Outro dos objetivos é de minimizar o tempo decorrido entre a ocorrência da anomalia, deteção e sua reparação [20].

Assim, este sistema deve ser capaz de identificar anomalias, falhas e erros dos componentes e subsistemas que têm um impacto nas emissões do motor do veiculo, (componentes ligados a uma *Engine control unit* (ECU)).

Alguns dos componentes e subsistemas que são monitorizados pela ECU de um veiculo:

- Sensores: Sensor de O2, sensores de temperatura, sensores de pressão;
- Atuadores: Injetores de combustível, bobinas de ignição;
- Sistema de injeção de combustível;
- Sistema de ignição.

O desenvolvimento do sistema OBDII trouxe uma normalização no que diz respeito a métodos de ligação e a nível de protocolos utilizados [21], dado que anteriormente, cada fabricante definia o seu próprio conector de ligação de diagnóstico, localização, especificações e procedimentos para comunicação com o veículo.

3.2.1. COMUNICAÇÃO

A especificação SAE J1962 prevê dois possíveis conectores, tipo A ou tipo B, de 16 pinos.

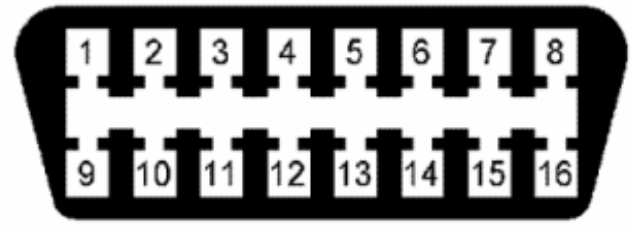


Figura 17 – Conector *standard* OBD [21]

O tipo A é utilizado em veículos com tensão de alimentação de 12V, enquanto o tipo B é utilizado em veículos com tensão de alimentação de 24V [24].

O SAE J1962 define o conector e respetivos pinos como ilustra a Figura 17 e Tabela 4.

Tabela 4 – Descrição dos pinos do conector standard OBD

Pino	Descrição
1	Definido pelo fabricante
2	SAE-J1850 *Bus positive Line*
3	Definido pelo fabricante
4	*Chassis ground*
5	*Signal ground*
6	*CAN High* (ISO15765-4 *e* SAE J2234)
7	*K-Line (*ISO9141-2 *e* ISO14230-4)
8	Definido pelo fabricante
9	Definido pelo fabricante
10	SAE-J1850 *Bus negative Line*
11	Definido pelo fabricante
12	Definido pelo fabricante
13	Definido pelo fabricante
14	*CAN Low* (ISO15765-4 *e* SAE J2234)
15	L-Line (ISO9141-2 *e* ISO14230-4)
16	*Battery voltage*

Apesar do conector ser padrão, para comunicar com os veículos podem ser utilizados vários protocolos de comunicação, dependendo do fabricante [23][24]:

- **ISO 9141-2**: Este protocolo é usado principalmente pela Chrysler, fabricantes europeus e asiáticos. É utilizada comunicação de forma serie assíncrona UART, com uma taxa de transmissão de 10.4 kbps, mensagens com tamanho de 5 a 11 bytes incluindo CRC, numa única linha bidirecional. Utiliza os pinos número 7 (*K-Line*) e número 15 (*L-Line* opcional) do conector J1962;
- **ISO 14230**: Tanto o protocolo como o *hardware* são quase idênticos ao do ISO 9141-2, com uma velocidade entre 1.2 e 10.4 kbps, tendo as mensagens um máximo de 255 bytes. São utilizados os pinos, os pinos 7 (*K-Line*) e 15 (*L-Line*, opcional) do conector SAE J1962;

- **SAE J1850 PWM (*Pulse Width Modulation*)** - O protocolo SAE J1850 PWM é o padrão utilizado pela *Ford Motors*, com o nome de *Standard Corporate Protocol* (SCP), mas isso não se aplica necessariamente a veículos *Ford* vendidos na Europa que usam o protocolo ISO. Possui taxa uma de transferência de 41.6 kbps, utiliza o pino 2 (sinal positivo) e pino 10 (sinal negativo) do conector SAE J1962. Utiliza uma tensão de 5V, e o tamanho de uma mensagem é de 12 bytes incluindo CRC, e emprega um sistema de controlo de acesso ao meio do tipo CSMA/CR;

- **SAE J1850 VPW (*Variable Pulse Width*)** – Este protocolo utiliza um único fio para comunicação, sendo considerado de baixo custo. É utilizado pela *General Motors*, com o nome de *GM Class 2*, e pela Chrysler com o nome de *Programmable Communication Interface* (PCI). Possui uma taxa de transferência de 10.4 kbps, utilizando apenas o pino 2 do conector SAE J1962 como sinal positivo utilizando uma tensão de 7V. O tamanho de uma mensagem é de 12 bytes incluindo CRC e emprega um sistema de controlo de acesso ao meio do tipo CSMA/CR;

- **ISO 15765** - O protocolo ISO 15765 é mais conhecido como *Controller Area Network* (CAN) e foi desenvolvido pela *Bosch*. Este é o protocolo que irá ser utilizado por todos os veículos, a partir de 2008, pelo menos nos Estados Unidos da América. Oferece a melhor rapidez e flexibilidade, com uma velocidade de até 500 kbps e utiliza o pino 6 (*CAN High*) e pino 14 (*CAN Low*) do conector SAE J1962.

3.2.2. MENSAGENS

O sistema OBD foi projetado de modo a este ser flexível, permitindo assim um meio para comunicação entre vários dispositivos. Esta comunicação é efetuada através de mensagens.

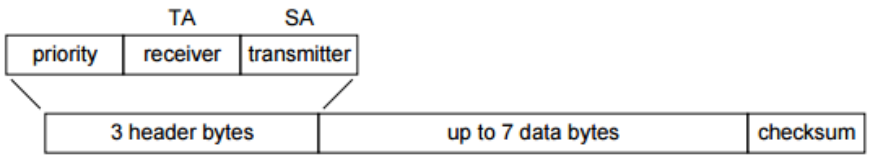


Figura 18 – Formato de mensagem OBDII (SAE J1850, ISO 9141-2 e ISO 14230) [26]

A Figura 18 ilustra a estrutura típica de uma mensagem utilizada pelo protocolo SAE J1850, ISO 9141-2 e ISO 14230.

Esta é composta pelos campos [26]:

- Cabeçalho de 3 bytes contendo a indicação da prioridade da mensagem, endereço de destino e endereço de origem;
- Campo de dados (até 7 bytes);
- Campo *Checksum* para verificação de erros.

No caso das mensagens do protocolo ISO 15765, é utilizada uma estrutura semelhante em que a principal diferença é a substituição do campo do cabeçalho por um campo ID de 11 ou 29bits consoante a versão do protocolo CAN utilizado, Figura 19.

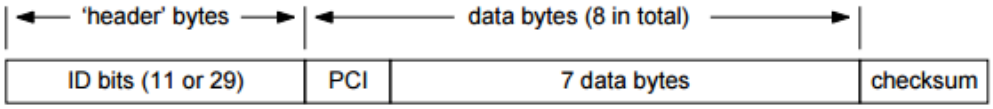


Figura 19 – Formato de mensagem OBDII (protocolo CAN) [26]

3.2.3. AQUISIÇÃO DE DADOS

As normas SAE J1979 e ISO 15031-5 definem como é efetuada a aquisição de dados entre o veículo e o equipamento de diagnóstico OBD.

Tabela 5 – Modos OBDII

Modo (HEX)	Descrição
01	Requisita dados atuais de um determinado PID.
02	Mostra dados de sensores do veículo na altura de uma avaria diagnosticada (*Freeze frame data*)
03	Requisita dos códigos de erro de diagnóstico
04	Elimina os códigos de erro armazenados na ECU
05	Requisita os resultados do teste de sensor de oxigênio
06	Requisita resultados dos testes de bordo de sistemas monitorizados
07	Requisita os códigos de erro detetados no ciclo de condução atual ou anterior (com impacto nas emissões)
08	Requisita controlo do sistema de bordo, teste ou componente
09	Requisita informações do veículo
0A	Requisita códigos de erro com impacto nas emissões

São definidos serviços ou modos, bem como parâmetros, denominados *Parameter IDentification* (PID), que permitem aceder a dados como a velocidade do veículo, rotação do motor, temperatura do líquido de refrigeração, entre outros.

Os serviços ou modos definidos pelas normas SAE J1979 e ISO 15031-5 são ilustradas na Tabela 5 [20]. Estes dados têm como origem a ECU do veículo [27]. Por estas normas alguns destes parâmetros são obrigatórios e outros opcionais, sendo permitido ainda a inclusão de outros parâmetros específicos do fabricante do veículo.

No que diz respeito aos PID, a norma *SAE J1979* define algumas dezenas de parâmetros possíveis [27]. A Tabela 6 descreve alguns exemplos dos parâmetros especificados na norma.

Tabela 6 – Exemplo de PID's especificados na norma SAEJ1979

PID (hex)	Dados devolvidos (bytes)	Descrição	Valor mínimo	Valor máximo	Unidades	Formula
05	1	Temperatura do líquido de refrigeração	-40	215	ºC	A-40
0C	2	Rotações do motor	0	16383.75	Rpm	((A*256) +B)/4
0D	1	Velocidade do veículo	0	255	Km/h	A
11	1	Posição do acelerador	0	100	%	A*100/255
0F	1	Temperatura do ar admitido pelo motor	-40	215	ºC	A-40

Uma falha num componente ou subsistema que tenha um impacto direto nas emissões do motor do veículo faz com que seja gravado na ECU do veículo, um código de erro ou *Diagnostic Trouble Code* (DTC), correspondente ao erro detetado.

Cada um destes códigos de erro é composto por uma letra seguida por quatro números como definido na norma SAE J2012. O código pode ser interpretado da forma ilustrada na Tabela 7 [28]. Assim um código de erro, por exemplo "P0351", representa uma falha da categoria "Motor ou transmissão", de um PID definido pelo ISO/SAE, no subsistema ou componente

número 3 e falha número 51. Os códigos de erro estão definidos no documento SAE J2012 que determina que o erro "P0351" é correspondente a *Ignition Coil "A" Primary/secundary circuit*.

Tabela 7 – Interpretação de DTC's [28]

Letra	P-Motor e transmissão (*Powertrain*)
	B-Carroçaria (*Body*)
	C-*Chassis*
	U-Rede
1º dígito	0 – Controlado pelo ISO/SAE
	1 – Controlado pelo fabricante
	2 - Controlado pelo fabricante
	3 - Controlado pelo fabricante/reservados pelo ISO/SAE
2º dígito	Subsistema ou componente
3º e 4º dígitos	Código de erro (00-99)

3.2.4. INTERFACES OBD

Atualmente estão disponíveis diversos sistemas que permitem a interação com o sistema de diagnóstico do veículo através da porta OBD [24].

Figura 20 – Dispositivo de leitura de DTC's

As suas funcionalidades dependem antes de mais, do seu preço e utilização. Podem ser simples dispositivos que permite apenas ler/apagar os DTC, Figura 20, ou dispositivos indicados para uso profissional, que permitem aceder a modos de diagnóstico avançado, aceder e alterar parâmetros específicos do fabricante do veículo, bem como aceder a módulos específicos, tais como o módulo do ar condicionado, ABS ou fecho centralizado.

Alguns destes dispositivos são completamente autónomos enquanto outros necessitam de um computador para funcionarem. Estes são conhecidos como adaptadores OBDII/USB, adaptadores sem fios OBDII/*Wi-Fi* ou OBDII/*Bluetooth*, que interagem com o utilizador através de *software* específico para computador ou telemóvel, como por exemplo: *VAG-COM Diagnostic System* (VCDS) que é um sistema de diagnóstico para veículos da marca *Volkswagen*, *Audi*, *Seat* e *Skoda*, Figura 21 e Torque (sistema de diagnostico para telemóveis com sistema operativo *Android*)

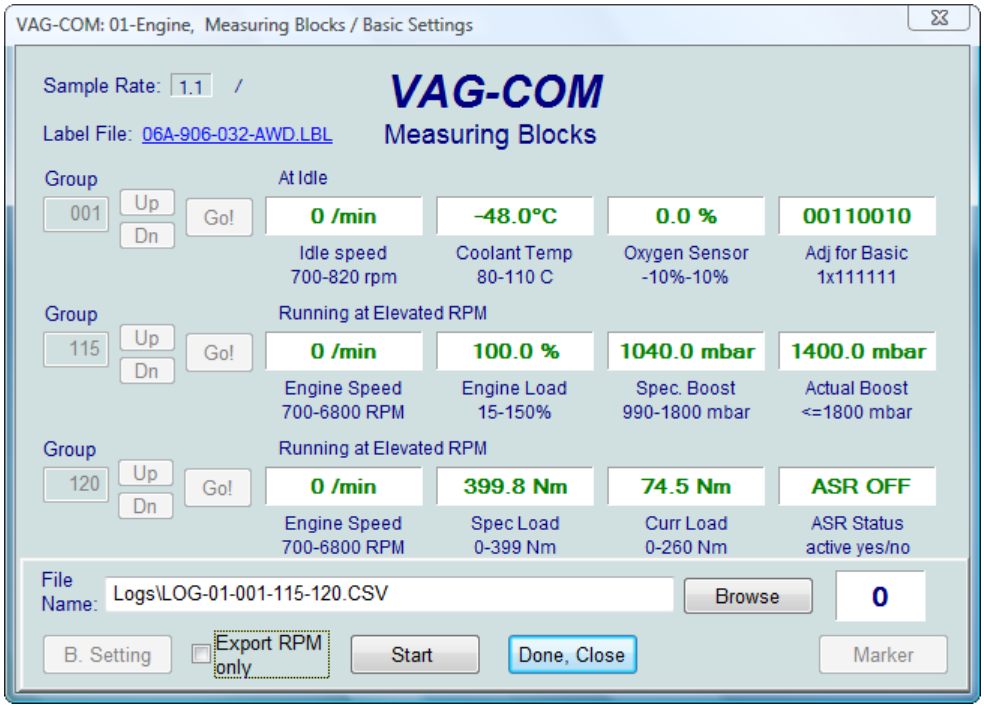


Figura 21 – *Software VAG-COM Diagnostic System (VCDS) [25]*

3.2.5. **ELM327**

O ELM327 é um microcontrolador programado pela *ELM Electronics* e é na sua essência um conversor de protocolos, neste caso OBDII/RS232.

Este *interface* é encontrado em adaptadores OBDII com e sem fios para uso com computadores, bem como em dispositivos de leitura de DTC's comerciais, tornando-se assim num dos mais populares conversores de protocolos OBDII. Uma das suas grandes vantagens é o baixo custo, bem como o facto de suportar praticamente todos os protocolos de comunicação OBD [26]. Por este facto será utilizado no desenvolvimento deste trabalho um adaptador/OBDII Bluetooth baseado no microcontrolador ELM327.

A comunicação com o ELM327 é efetuada através de comandos AT, que se encontram listados na documentação do fabricante [26].

A Tabela 8 ilustra alguns comandos AT e sua função, para uso com o ELM327.

Tabela 8 – Exemplo de comandos AT (ELM327)

Comando	Descrição
ATZ	Faz reset ao ELM327
AT SP 0	Seleciona o protocolo OBD automaticamente
AT E0 ou AT E1	Impede que o comando enviado para o ELM327 seja enviado na resposta (echo)
ATI	Devolve a versão do ELM327
<CR>	Repete o último comando

Para comunicar com um adaptador OBDII baseado no ELM327, pode ser utilizado um programa de comunicação como o *Hyperterminal* do *Microsoft Windows* configurando a porta serie de um computador da seguinte forma:

- *Bits per second: 9600bps*
- *Data bits: 8*
- *Parity: None*
- *Stop bits: 1*
- *Flow control: None*

Após esta configuração é possível por exemplo, obter a temperatura do líquido de refrigeração do motor do veículo efetuando os seguintes passos:

1. Envio do comando "AT *Z*" - *Reset* ao ELM327
2. Resposta do ELM327 (versão do chip) - "ELM327 v1.3>"

3. Envio do comando “AT SP 0” – Seleciona o protocolo OBD automaticamente.
4. Resposta do ELM327 “OK”
5. Envio do comando “01 05” – Requisita a leitura do sensor do líquido de refrigeração do motor (Modo 01, PID 05)
6. Resposta do ELM327 - “41 05 7B>”

Para interpretar a resposta do ELM327 é necessário, antes de mais, saber qual o formato da sua resposta. A Figura 22 ilustra o formato de uma resposta do ELM327 a um pedido efetuado pelo utilizador.

1 byte	1 byte	variável (depende do pid)
0x40 + modo	pid	dados

Figura 22 – Resposta do ELM327

Como é possível verificar na resposta (41 05 7B>) do ELM327 ao envio do comando “01 05”, o primeiro byte indica que a resposta é a um pedido no modo 01 (0x40 + 01=0x41). O segundo byte representa o número do PID requisitado. Assim, “41 05” representa uma resposta a um pedido no modo 01 e PID 05, tal como por exemplo, “43 01”, representaria uma resposta a um pedido no modo 03, PID 01. Já o último byte (7B) representa a leitura do sensor em hexadecimal, que após conversão do valor para decimal e aplicando a fórmula ilustrada na Tabela 6, o valor de temperatura em graus Celcius devolvido pelo sensor é de 83ºC ((11 + 7*16)-40). É de realçar que a Tabela 6 indica que a resposta ao PID “05” devolve apenas um *byte*. O último caracter “>” indica que o dispositivo está pronto a receber novas instruções.

3.3. O SISTEMA OPERATIVO ANDROID

O *Android* é um sistema operacional baseado no *kernel* do Linux e atualmente desenvolvido pela empresa Google. O sistema foi inicialmente desenvolvido pela *Android Inc*., empresa posteriormente adquirida pelo Google em 2005. A Figura 23 ilustra um exemplo do ecrã inicial e menu do sistema operativo *Android*. O Google anunciou o *Android* e a *Open*

Handset Alliance (OHA) em 2007, uma aliança 84 empresas, com o objetivo de criar padrões abertos para dispositivos móveis [29].

Figura 23 – Exemplo de um ecrã inicial e menu do sistema operativo *Android*

A OHA conta com alguns dos maiores fabricantes de dispositivos móveis, operadoras de telecomunicações, fabricantes de semicondutores e empresas de *software*, como por exemplo: Google, *HTC, Sony, Dell, Intel, Motorola, Qualcomm, Wind River Systems, Nvidia, Texas Instruments, Samsung Electronics, LG Electronics e T-Mobile,Sprint Corporation.*

O *Android* foi idealizado principalmente para dispositivos móveis com *touchscreen*, tal como *smartphones* e *tablets*, com *interfaces* de utilizador especializados para televisores (*Android* TV), automóveis (*Android* Auto), e relógios de pulso (*Android Wear*). Tem sido utilizado também em consolas de jogos, algumas câmaras fotográficas digitais e outros aparelhos eletrónicos.

3.3.1. A PLATAFORMA ANDROID

O *Android* é sistema baseado no *kernel Linux*, e composto por quatro camadas principais [30] como ilustra a Figura 24:

- *Application Layer* - Todas as aplicações (nativas ou criadas por terceiros) estão nesta camada, tal como a aplicação *Phone* e *Browser*;

- *Application Framework* – A *Application Framework* fornece as classes utilizadas para a criação das aplicações *Android*, fornecendo uma abstração para acesso ao *hardware*. Por exemplo: *Location Manager*, *Window Manager e Activity Manager*;

- *Libraries* - Bibliotecas escritas em C/C++ que correm por cima do *Linux Kernel*;

- *Linux kernel* - Interação com o *hardware* através de *drivers* de dispositivos.

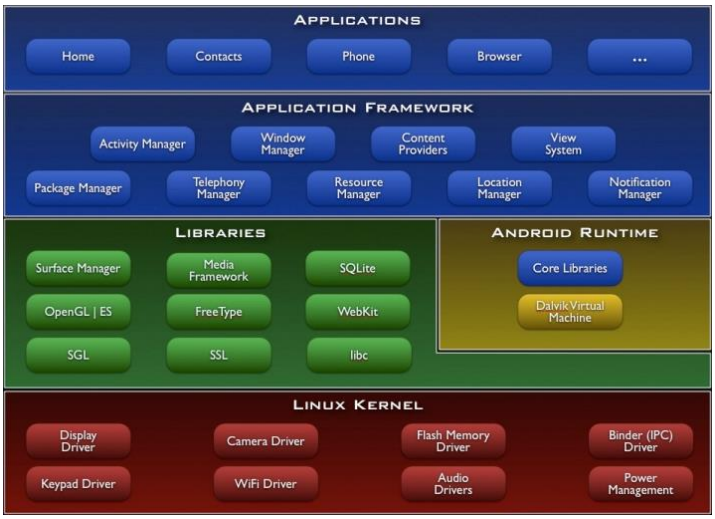


Figura 24 – *Android Framework* [30]

3.3.2. ANDROID SOFTWARE DEVELOPMENT KIT

As aplicações são geralmente desenvolvidas na linguagem de programação Java utilizando o *Android Software Development Kit* (SDK). O SDK inclui um conjunto de ferramentas de desenvolvimento incluindo um depurador, bibliotecas, um emulador, documentação, exemplos e tutoriais.

O *Integrated Development Environment* (IDE) oficialmente suportado é o *software Android Studio*, embora outros possam ser utilizados, como o *Eclipse*, *IntelliJ IDEA* IDE ou o *NetBeans* IDE.

3.3.3. ARQUITETURA DAS APLICAÇÕES

Conforme mencionado anteriormente, o *Android* é executado sobre um *kernel Linux*. Os aplicativos *Android* são executados numa máquina virtual (*Dalvik Virtual Machine*), uma tecnologia de *software* livre. Cada um dos aplicativos *Android* é executado numa instância

da *Dalvik VM*, que, por sua vez, reside num processo que é gerido pelo *kernel Linux*, conforme ilustra a Figura 25 [30].

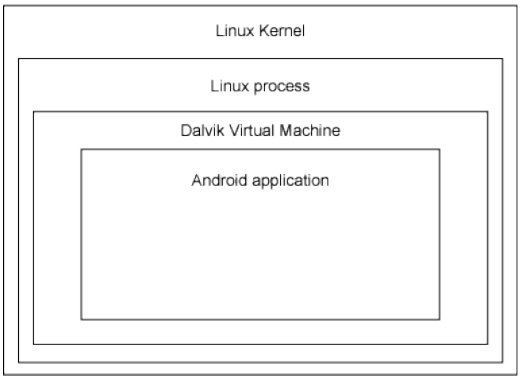


Figura 25 – Execução de uma aplicação *Android [30]*

Componentes das Aplicações [31]:

Ativities - Uma *Activity* ou atividade, representa um ecrã da *user interface* (UI). Uma aplicação que tem uma UI visível ao utilizador é implementado com uma atividade. As aplicações podem ser constituídas por mais do que uma atividade dependendo do número de diferentes ecrãs ou funcionalidades que possuir. Alguns tipos de atividades*: Activity, ListActivity e AppCompatActivity*.

Uma aplicação *Android* geralmente consiste em múltiplas atividades ligadas entre si. Cada atividade pode iniciar uma outra atividade, a fim de executar ações diferentes. Quando uma atividade é interrompida porque uma nova atividade é iniciada, esta é notificada dessa mudança de estado através de métodos. Existem vários métodos de retorno que uma atividade pode receber, como visível na Figura 26.

- *onCreate*() - O primeiro evento utilizado na *Ativity*, no momento da sua criação;
- *onStart*() - *Activity* é iniciada;
- *onResume*() -*Activity* é visível ao utilizador;
- *onPause*() - Chamado quando o sistema está prestes a começar a retomar outra atividade. A *Activity* deixa de ser visível ao utilizador;
- *onStop*() - A *Activity* é parada;
- *onRestart*() - Chamado quando uma atividade parou por algum motivo. Executando o *OnStart*() de uma forma automática;

- *onDestroy*(): Quando é atividade é destruída.

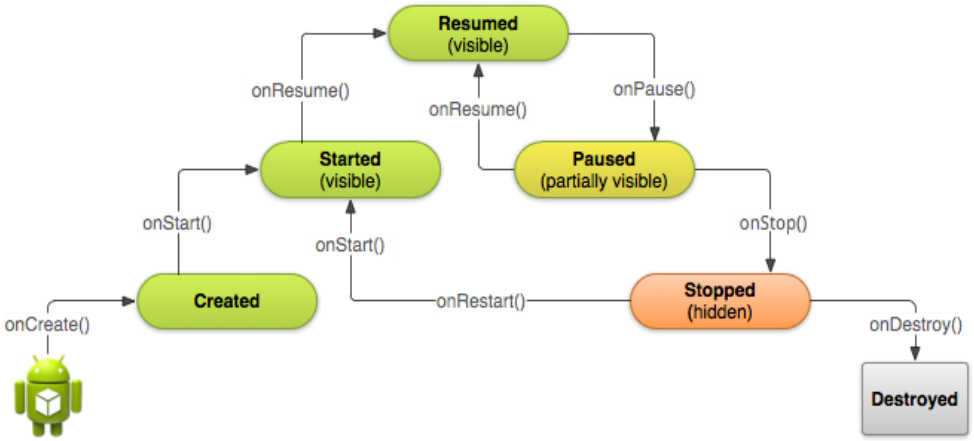


Figura 26 – Ciclo de vida de uma Activity [31]

Fragments - Um fragmento, disponível a partir da versão 3.0 (versão API11) do *Android* é quase uma atividade dado que possui o seu ciclo de vida próprio. Tem algumas vantagens, tais como, aumentar o reúso de atividades (já que é possível ter múltiplos fragmentos na mesma atividade), permitem a construção de ecrãs com múltiplas visões e também a modificação da visão de uma atividade em tempo de execução.

Services - Um serviço é uma tarefa executada em *background*. É normalmente utilizado em aplicações que necessitam de uma execução por um longo período de tempo, como uma aplicação de verificação de atualizações, um leitor de música, entre outros. Basicamente, existem dois tipos de serviços:

- *Unbounded*: Tipo de serviço completamente independente da atividade que o iniciou, não existindo comunicação ou interação entre eles.

- *Bounded*: Tipo de serviço que oferece comunicação e interação entre este e a atividade que a iniciou. Utilizando mensagens e *BroadcastReceivers* a atividade pode a qualquer momento monitorizar o estado ou o progresso da tarefa que o serviço está a realizar, bem como permitir a trocar dados.

Content Providers – O *Content Provider* é uma API que permite disponibilizar dados a outras aplicações ou atividades. Como exemplo o *Android* possui um *ContentProvider* para os contactos (nomes, endereços, número de telefone, etc.), permitindo assim que estes possam ser acedidos por qualquer aplicação.

Intents - Os *Intents* são usados como um mecanismo de passagem de mensagens entre aplicações permitindo descrever uma ação, como por exemplo, "tirar uma fotografia", "ligar para casa" ou "mostrar uma atividade". Cada *Intent* está associado a um componente, o qual é invocado sempre que o seu *Intent* é chamado; Exemplo: Para enviar um *email* a partir de uma aplicação pode-se invocar o *Intent* "enviar um email". Este abrirá a atividade que envia *emails* por predefinição no sistema.

Broadcast Receivers **-** Um aplicativo *Android* pode ser ativado para processar um elemento de dados ou para responder a um dado evento, como a receção de uma mensagem de texto, hora do sistema alterada, entre outros.

AndroidManifest.xml **–** Cada aplicação *Android* possui um ficheiro AndroidManifest.xml. A função deste ficheiro é expor informações sobre a aplicação ao sistema *Android*. O *AndroidManifest* possui, por exemplo, a seguinte informação:

- *Package Java* da aplicação;
- Descreve quais os componentes que a aplicação possui: *User Interfaces (Activity), Service, Broadcast Receivers* e *Content Providers;*
- Descreve quais as permissões que a aplicação necessita, ex.: acesso à internet, acesso ao armazenamento externo;
- Declara qual a versão do SDK mínima, máxima e a utilizada;
- As funcionalidades utilizadas (câmara, autofócus, etc.)

3.3.4. SERVIÇO DE LOCALIZAÇÃO

A maioria dos dispositivos *Android* permitem determinar a sua localização geográfica atual. Isto pode ser efetuado através da utilização de um módulo GPS (normalmente interno), rede GSM ou através de redes WI-FI.

O *Android* fornece um pacote denominado *android.location* que fornece acesso aos serviços de localização suportados pelo dispositivo através de uma série de classes, sendo o seu principal componente, o serviço *LocationManager [32]*.

Este serviço disponibiliza APIs para determinar a localização, registar *event listeners* para atualização da localização, alertas de proximidade, entre outros.

Obter a localização do dispositivo funciona através de um *callback*. É necessário solicitar ao *LocationManager* o envio de atualizações de localização através do método *requestLocationUpdates(),* passando-lhe um *LocationListener* que basicamente é um *interface.* Este *LocationListener* deve implementar vários métodos que o *Location Manager* chama quando a localização ou *status* do dispositivo sofre alterações. O código seguinte demonstra como definir um *LocationListener* e efetuar um pedido de localização:

```
   // Location Manager
LocationManager locationManager = (LocationManager)
this.getSystemService(Context.LOCATION_SERVICE);

   // Location Listener
LocationListener locationListener = new
LocationListener() {

   // Caso a localização seja alterada
   public void onLocationChanged(Location location) {
         //
         // Código... }

   // Caso o estado do serviço de localização seja
alterado
   public void onStatusChanged(String provider, int
status, Bundle extras) {}

   // Caso o estado do serviço de localização seja
activado
   public void onProviderEnabled(String provider) {}

   // Caso o estado do serviço de localização seja
desactivado
   public void onProviderDisabled(String provider) {}
     };

   // Registar o listener com o locationManager de
modo a receber actualizações de localização

   locationManager.requestLocationUpdates(LocationMana
ger.NETWORK_PROVIDER, 0, 0, locationListener);
```

O primeiro parâmetro do *requestLocationUpdates()* é o tipo de fornecedor de localização a utilizar.

requestLocationUpdates (String provider, long minTime, float minDistance, LocationListener listener)

O *provider* pode ser do tipo "GPS_PROVIDER" (utiliza GPS para determinar a localização) ou "NETWORK_PROVIDER" (determina a localização com base em redes GSM e WI-FI) consoante a necessidade ou o que é pretendido. A frequência de receção das atualizações pode ser controlada utilizando o segundo e terceiro parâmetro (*minTime e minDistance*),

sendo o segundo parâmetro o intervalo de tempo mínimo entre notificações e o terceiro a distância mínima percorrida pelo dispositivo entre as notificações. Colocando os dois parâmetros a zero, as notificações são recebidas tão frequentemente quanto possível. O último parâmetro é o *LocationListener*, cujo método *onLocationChanged(Location)* será chamado a cada atualização de localização. De modo a receber estas notificações é necessário adicionar ao ficheiro AndroidManifest.xml as permissões necessárias:

- ACCESS_FINE_LOCATION - Permite à aplicação aceder à localização através de GPS, e redes móveis.

```
  <uses-permission android:name="
android.permission.ACCESS_FINE_LOCATION" />
```

Uma localização pode ser constituída pela latitude, longitude, data e hora, e outras informações tais como altitude e velocidade. Todas as localizações geradas pelo *LocationManager* têm latitude, longitude e *timestamp* válidos. De modo aceder diretamente aos valores obtidos pelo *LocationManager* é necessário aceder à classe *Location* (uma classe de dados que representa uma localização geográfica) através de determinados métodos, tais como:

- *getAccuracy*() – Devolve a precisão estimada da localização, em metros;
- *getLatitude*() – Latitude, em graus;
- *getLongitude*() – Longitude, em graus;
- *getProvider*() – Devolve o nome do *provider* que forneceu a localização;
- *getSpeed*() – Velocidade (se disponível) em metros por segundo;
- *getTime*() - UTC time da localização, em milissegundos desde 01/01/1970;

3.3.5. INTERAÇÃO COM DISPOSITIVOS *BLUEOOTH*

O *Android* fornece um pacote denominado *android.bluetooth* que fornece acesso às *Bluetooth* API's. Utilizando estas APIs uma aplicação *Android* pode [33]:

- Procurar dispositivos *Bluetooth;*

- Determinar quais os dispositivos já emparelhados;
- Estabelecer canais RFCOMM;
- Descobrir outros dispositivos através do *Service Discovery;*
- Transferir dados de e para outros dispositivos;
- Gerir múltiplas ligações.

De forma a criar ligações *Bluetooth* é necessário utilizar uma serie de classes, tais como:

- *BluetoothAdapter* - Representa o módulo *Bluetooth* do dispositivo. O *BluetoothAdapter* é o ponto de entrada para todas as atividades com *Bluetooth*. Utilizando-o é possível consultar uma lista de dispositivos emparelhados, instanciar um dispositivo *Bluetooth* utilizando um endereço MAC conhecido, entre outros;
- *BluetoothDevice* - Representa um dispositivo *Bluetooth* remoto. É utilizado para solicitar uma ligação com um dispositivo remoto através de um *BluetoothSocket* ou consultar informações sobre o dispositivo, como o nome, endereço, classe ou estado.
- *BluetoothSocket* - Representa o interface para um *socket Bluetooth*. Este é o ponto de ligação que permite a uma aplicação trocar dados com um outro dispositivo *Bluetooth*.

De forma a utilizar o *Bluetooth* numa aplicação é necessário adicionar ao ficheiro *AndroidManifest.xml* as permissões necessárias:

- BLUETOOTH – Permite a comunicação *Bluetooth*, ou seja, solicitar ou aceitar uma ligação, bem como a transferência de dados;
- BLUETOOTH_ADMIN – Permite iniciar a pesquisa de dispositivos ou alterar as definições de *Bluetooth*.

O código introduzido no ficheiro *AndroidManifest.xml* é o seguinte:

```
      <uses-permission
android:name="android.permission.BLUETOOTH"/>

      <uses-permission
android:name="android.permission.BLUETOOTH_ADMIN"/>
```

Para iniciar uma comunicação *Bluetooth* é necessário obter um *BluetoothAdapter [33]*. Para o obter invoca-se o método *getDefaultAdapter() que retorna* o *BluetoothAdapter*.

No caso de o método retornar *Null*, isto significa que o dispositivo não suporta *Bluetooth*.

```
   // Obter o BluetoothAdapter
   BluetoothAdapter mBluetoothAdapter =
BluetoothAdapter.getDefaultAdapter();

   if (mBluetoothAdapter == null)
   { // O dispositivo não suporta Bluetooth}
```

De seguida torna-se necessário verificar se o *Bluetooth* está ativo. O método *isEnabled()* irá retornar *"true"* ou *"false"*, consoante o estado do *Bluetooth*. Para solicitar a ativação do *Bluetooth* chama-se o método *startActivityForResult()* com o *Intent* "ACTION_REQUEST_ENABLE".

```
   // Bluetooth desactivado
   if (!mBluetoothAdapter.isEnabled())
   {
       Intent enableBtIntent = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);

       startActivityForResult(enableBtIntent,
REQUEST_ENABLE_BT); }
```

Isto irá emitir um pedido de ativação do *Bluetooth* através das configurações do sistema e mostrar uma *dialogbox*, Figura 27, que solicita ao utilizador a permissão para ativar o *Bluetooth* caso este esteja desativado.

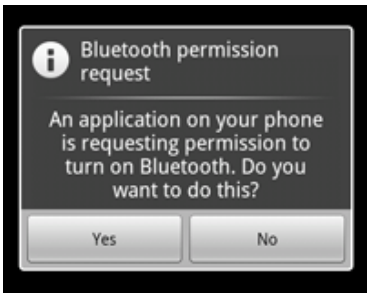


Figura 27 – Pedido de activação do *Bluetooth*

De forma a iniciar uma ligação com outro dispositivo *Bluetooth*, como cliente, é necessário obter um objeto *BluetoothDevice* que irá representar o dispositivo remoto.

A obtenção deste objeto pode ser efetuada, por exemplo, com recurso ao método *mBluetoothAdapter.getBondedDevices()*, que irá retornar um "*set"* de dispositivos emparelhados.

De seguida é necessário um *BluetoothSocket*, através da chamada ao método *createRfcommSocketToServiceRecord()*, que inicializa um *BluetoothSocket* que irá conectar-se ao *BluetoothDevice*.

Com este *BluetoothSocket* e chamando o método *connect()*, no caso de sucesso na ligação este irá retornar. No caso de insucesso e dado que este método é bloqueante, uma exceção será disparada.

O método *createRfcommSocketToServiceRecord()* requer como parâmetro um *Universally Unique IDentifier* (UUID). O UUID é utilizado para identificar um serviço particular e perfil correspondente fornecido por um dispositivo *Bluetooth*.

```
  private class ConnectThread extends Thread {
      private final BluetoothSocket mmSocket;
      private final BluetoothDevice mmDevice;

  public ConnectThread(BluetoothDevice device) {

      BluetoothSocket tmp = null;
      mmDevice = device;

  try {     // Obtenção do BluetoothSocket para
ligação a um dado dispositivo Bluetooth

tmp = device.createRfcommSocketToServiceRecord
(MY_UUID); }

  (...)
  public void run() {
    try {
              // Ligar ao dispositivo
              mmSocket.connect();}

    catch (IOException connectException) {
     // Não foi possível a ligação
      try {
              mmSocket.close();
          } catch (IOException closeException) {}
              return;
          }

          // Gestão da ligação criada
          manageConnectedSocket(mmSocket);
    (...)
```

Após a ligação efetuada entre dois ou mais dispositivos, é necessário obter um *InputStream* e *OutputStream*, com recurso aos métodos *getInputStream()* e *getOutputStream()*

respetivamente, de modo a manipular as transmissões através do *socket* criado, como ilustra o seguinte exemplo:

```
  private class ConnectedThread extends Thread {
      private final BluetoothSocket mmSocket;
      private final InputStream mmInStream;
      private final OutputStream mmOutStream;

      public ConnectedThread(BluetoothSocket socket)
{
          mmSocket = socket;
          InputStream tmpIn = null;
          OutputStream tmpOut = null;

  try {
        // Input e OutputStream
        tmpIn = socket.getInputStream();
        tmpOut = socket.getOutputStream();
      } catch (IOException e) {}

        mmInStream = tmpIn;
        mmOutStream = tmpOut;
      }
```

Posteriormente é possível escrever e ler dados através dos métodos *read(byte[])* e *write(byte[])*. O seguinte exemplo ilustra a utilização de uma *thread* para troca de dados com um dispositivo remoto *Bluetooth*.

```
  public void run() {

  byte[] buffer = new byte[1]; // Stream Buffer
  int bytes; // bytes devolvidos pelo método read

  while (true) {
  try {    // Lêr dados do InputStream

  bytes = mmInStream.read(buffer);

  // Envia dados obtidos para a Atividade (UI)
  mHandler.obtainMessage(MESSAGE_READ, bytes, -1,
buffer).sendToTarget();}

        catch (IOException e){ break; }   }}

  public void write(byte[] bytes) {
  try {// Envia dados para o dispositivo remoto

  mmOutStream.write(bytes);
   } catch (IOException e) {}
```

3.4. PHP E MYSQL

De modo a enviar e receber dados do dispositivo *Android* para um servidor Web será utilizada uma API que utiliza as tecnologias PHP e MYSQL.

3.4.1. MYSQL

O MySQL é um sistema de gestão de base de dados [34], que utiliza a linguagem SQL como interface. Tem como principais características:

- Executado num servidor (*server-sided*)
- Boa performance, fiável e fácil de utilizar
- Ideal tanto para pequenas como grandes aplicações
- Utiliza linguagem SQL *standard*
- Disponível para um grande número de plataformas
- Gratuito

A Tabela 9 descreve alguns dos comandos SQL utilizados para interagir com os dados contidos na base de dados.

Tabela 9 – Exemplo de alguns comandos SQL

Categoria	Comando	Descrição
Consulta de dados	SELECT	Utilizado para selecionar determinados registros de uma ou mais tabelas.
Manipulação de Dados	INSERT	Utilizado para criar um registro
	UPDATE	Utilizado para alterar certos registros.
	DELETE	Utilizado para excluir determinados registros
Definição de Dados:	CREATE	Utilizado para criar uma nova tabela ou outro objeto
	ALTER	Utilizado para modificar um objeto existente, como uma tabela
	DROP	Utilizado para apagar uma tabela inteira, ou outro objeto

Exemplo de um código SQL que seleciona todos os registos de uma tabela “locations” que contenha o campo “uid” igual a um determinado valor:

```
SELECT * FROM locations WHERE uid = $uid
```

3.4.2. **PHP**

O PHP é uma linguagem de programação de uso geral que é geralmente muito utilizada para criar páginas Web dinâmicas, sendo que o código é executado normalmente num servidor (*server-sided*).

Permite por exemplo avaliar dados de um formulário enviados por um *browser* e contém módulos para interagir com base de dados, servidores FTP, entre outros.

O código seguinte demonstra como o PHP pode facilmente interagir com uma base de dados Mysql.

```
<?php
// Dados de acesso à base de dados
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';

// Ligar à base de dados
$conn = mysql_connect($dbhost, $dbuser, $dbpass);

// Se a ligação não for possível
if(!$conn)
{die('Ligação falhou: ' . mysql_error());}

// Criar um pedido para a criação de uma Tabela
$sql = "CREATE TABLE TESE( ".
   "ID_ALUNO INT NOT NULL AUTO_INCREMENT, ".
   "NUMERO INT NOT NULL, ".
   "PRIMEIRO_NOME VARCHAR(20) NOT NULL, ".
   "ULTIMO_NOME VARCHAR(20) NOT NULL, ".
   "DATA DATE, ".
   "PRIMARY KEY (ID_ALUNO)); ";

// Escolha da base de dados e criação da tabela
mysql_select_db( 'TESE' );
mysql_query( $sql, $conn );

// Fechar ligação   com a base de dados
mysql_close($conn);
?>
```

Para aceder à base de dados é necessário definir os dados de acesso bem como o endereço e porta do servidor. O método *mysql_connect()* efetua a ligação ao servidor e posteriormente é feita uma verificação do estado da ligação. Para a criação de uma tabela utiliza-se o comando SQL “CREATE TABLE”. Neste caso é criada uma tabela “TESE” tendo como campos o “id”, número, primeiro e ultimo nome e a data, como ilustra a Tabela 10. A *Primary Key* identifica o campo que contem valores únicos e que não se repetem (neste caso

o campo “id” do aluno). Esta chave apenas pode conter valores únicos e deve sempre conter um valor.

Tabela 10 – Tabela Mysql criada

Campo	Tipo de dados	Outros parâmetros	
ID	INT	NOT NULL	AUTO_INCREMENT
NUMERO	INT	NOT NULL	-
PRIMEIRO_NOME	VARCHAR(20)	NOT NULL	-
ULTIMO_NOME	VARCHAR(20)	NOT NULL	-
DATA	DATE	-	-

Cada campo da tabela MYSQL pode ser descrito da seguinte forma:

- **ID** – Campo com dados do tipo inteiro, não nulo e que se auto incrementa de modo a cada registo ter um valor único;
- **NUMERO** – Campo com dados do tipo inteiro, não nulo;
- **PRIMEIRO_NOME** – Campo de vinte caracteres, não nulo;
- **ULTIMO_NOME** – Campo de vinte caracteres, não nulo;
- **DATA** – Campo do tipo DATE, no formato 'AAAA-MM-DD'

A Tabela 11 ilustra o exemplo de um registo criado na base de dados na tabela TESE.

Tabela 11 – Registo criado na base de dados

ID_ALUNO	NUMERO	PRIMEIRO_NOME	ULTIMO_NOME	DATA
1	1070253	José	Pina	2015-08-15

3.4.3. API PARA SISTEMA DE LOGIN/REGISTO

Para a troca de dados entre a aplicação *Android* e uma base de dados Mysql será utilizada um API desenvolvida por *Ravi Tamada*, de modo a implementar um sistema de login e registo para a aplicação [35]. A Figura 28 ilustra a sua arquitetura geral.

Esta API tem como características:

- Aceitar pedidos através dos métodos GET ou POST;
- Ligação com base de dados MYSQL através de PHP;
- Respostas do servidor em formato JSON.

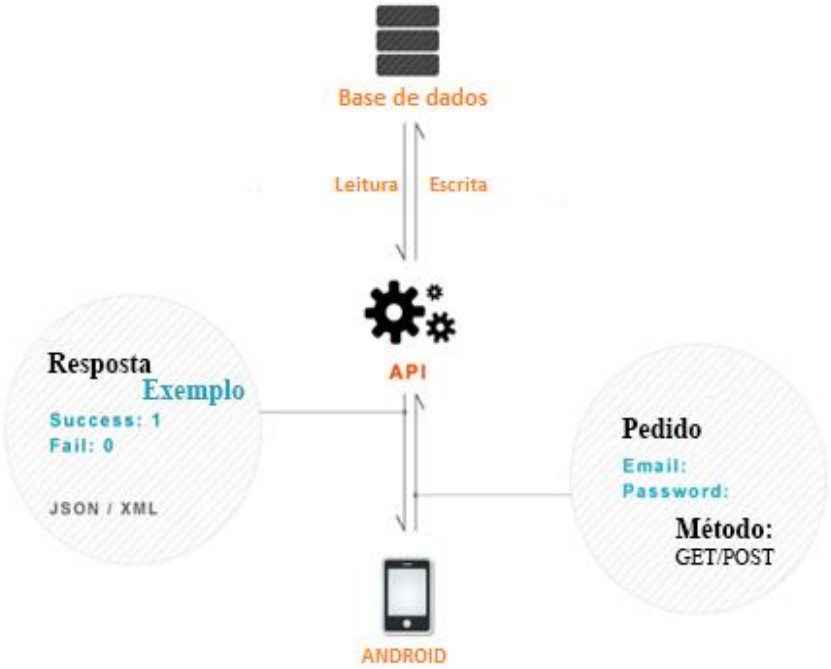


Figura 28 – Arquitetura da API para sistema de *login*/registo [35]

A API é composta por um conjunto de ficheiros PHP (*server side*) e Java (*cliente side*). No servidor estão alojados quatro ficheiros PHP: *Config, DB_connect, DB_functions e Index.* A Figura 29 ilustra o fluxo de dados da API no que diz respeito à aplicação em PHP.

Estes ficheiros possuem as seguintes funções:

- *Config.php* – Contém dados de acesso à base de dados, tal como o *username e password;*
- *DB_connect.php* – Classe que contém os métodos de ligação à base de dados;
    - *connect*() – Efetua ligação;
    - *close*() – Termina ligação.

- *DB_functions.php* – Classe que define métodos de interação (leitura e escrita) com a base de dados;
    - *storeUser*() – Regista um utilizador, retorna os dados do utilizador em caso de sucesso;
    - *getUserByEmailAndPassword*() – Login – Retorna utilizador, dado o email e *password*;
    - *isUserExisted*() - Verifica se o utilizador já existe (no caso de novo registo);
    - *hashSSHA*() – Efetua a encriptação da *password*. Retorna o *salt* e a *password*;
    - *checkhashSSHA*() – Efetua a desencriptação da *password*. Retorna uma *hash string*.
- *index.php* – Aceita pedidos de ligação GET e POST e envia notificação em formato JSON, que é um formato leve para troca de dados computacionais.

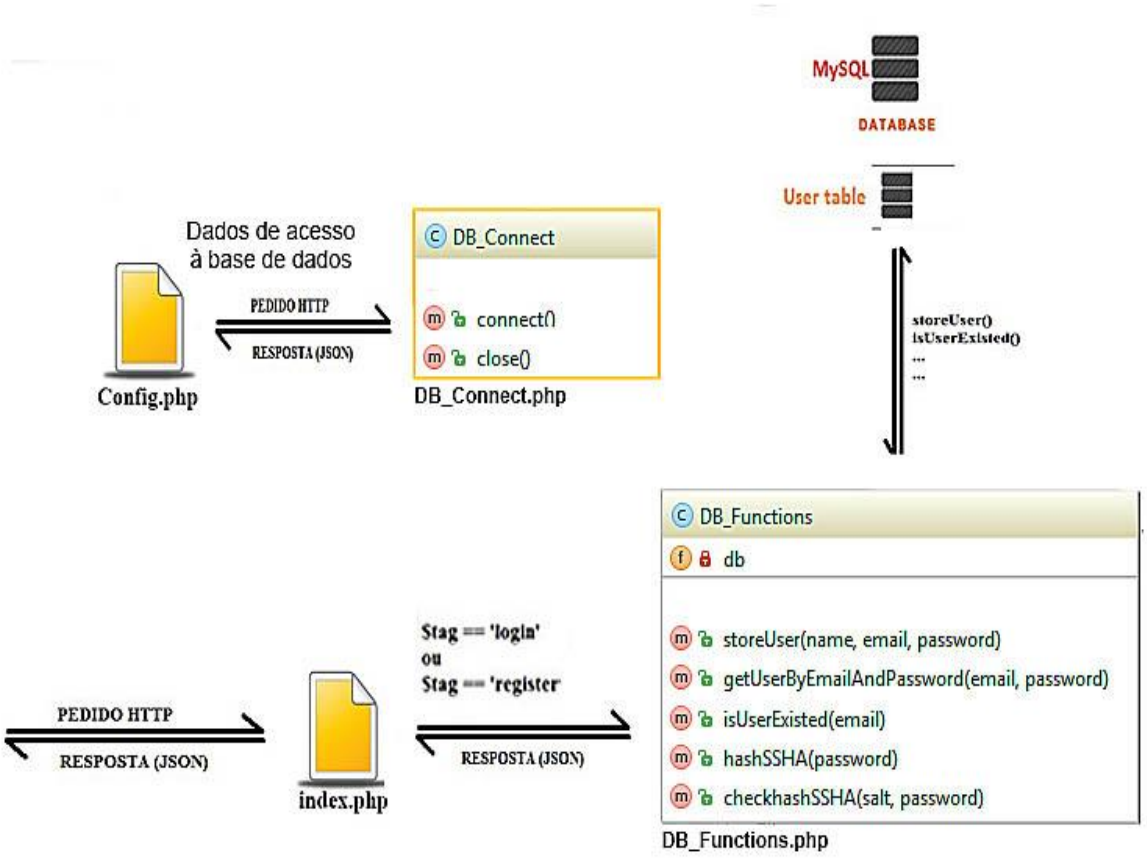


Figura 29 – Fluxo de dados da API (aplicação PHP)

Tanto no modo de registo como de login, são enviadas respostas como comprovativo das operações.

O seguinte código ilustra a resposta recebida em formato JSON, a um registo de utilizador completado com sucesso.

```
{
    "tag": "register",
    "success": 1,
    "error": 0,
    "uid": "4f074ca1e3df49.06340261",
    "user": {
        "name": "Jose Pina",
        "email": "email@gmail.com",
        "created_at": "2015-05-07 01:03:53",
        "updated_at": null
    }
}
```

A Tabela 12 e Tabela 13 ilustram, respetivamente, o tipo de mensagens recebidas e códigos de erro no modo de registo e login.

Tabela 12 – Notificações em modo de registo

Notificações (modo de registo)	**Variáveis**	
Registado com sucesso	*success*=1	*error*=0
Erro no registo	*success*=0	*error*=1
Nome de utilizador já existe	*success*=0	*error*=2

Tabela 13 – Notificações em modo de login

Notificações (modo de login)	**Variáveis**	
Login OK	*success*=1	*error*=0
Login Error	*success*=0	*error*=1

Para a aplicação *Android*, a API disponibiliza três classes, *JSON Parser Class, SQLite Database Handler Class* e *User Functions Class*. O diagrama de classes da API é ilustrado na Figura 30.

As funcionalidades destas classes são as seguintes:

- *JSON Parser Class* - Esta classe permite interpretar os dados de resposta do servidor em formato JSON. Assim, é utilizada para determinar o tipo de método a ser usado, POST ou GET efetuando um pedido HTTP e obtém a resposta em formato JSON;

- *User Functions Class* - Esta classe contém métodos para lidar com todos os eventos do utilizador:

    - *loginUser*() – Efetua um pedido de *login*, tendo como parâmetros o *email* e *password* do utilizador;

    - *registerUser*() - Efetua um pedido de registo, tendo como parâmetros o nome, *email* e *password* do utilizador;

    - *isUserLoggedIn*() – Reporta o estado do utilizador (*login*);

    - *logoutUser*() – Efetua o *logout* do utilizador.

- *SQLite Database Handler Class* – Esta classe permite armazenar informações do utilizador no dispositivo, nomeadamente após o login e registo de utilizador, utilizando uma base de dados SQLite. O sistema *Android* possui suporte nativo para o SQLite que é uma pequena biblioteca, desenvolvida em linguagem C, que implementa um motor de base de dados com a interface dentro de uma única biblioteca. A Tabela 14 ilustra a tabela criada pela classe.

Tabela 14 – Tabela criada pela classe *SQLite Database Handler*

Campo	Tipo de variável	Primary Key
Id	INT	PRI
Nome	TEXT	-
Email	TEXT	-
Uid	TEXT	-
Criado em	TEXT	-

Esta tabela contem o "Id", nome, *email*, "uid" e data de criação do utilizador. Assim sendo, esta classe tem como função criar e interagir com a base de dados SQLite, implementando métodos de escrita e leitura:

- *onCreate*() – Cria a tabela necessária;
- *onUpgrade*() – Efetua um upgrade à base de dados;
- *addUser*() – Adiciona detalhes do utilizador;
- *getUserDetails*() – Lê os detalhes do utilizador;

- *getRowCount*() – Estado do utilizador (login);
- *resetTables*() – Apaga dados da tabela.

Esta API fornece um exemplo das atividades de login e registo, sendo estas utilizadas no desenvolvimento deste trabalho.

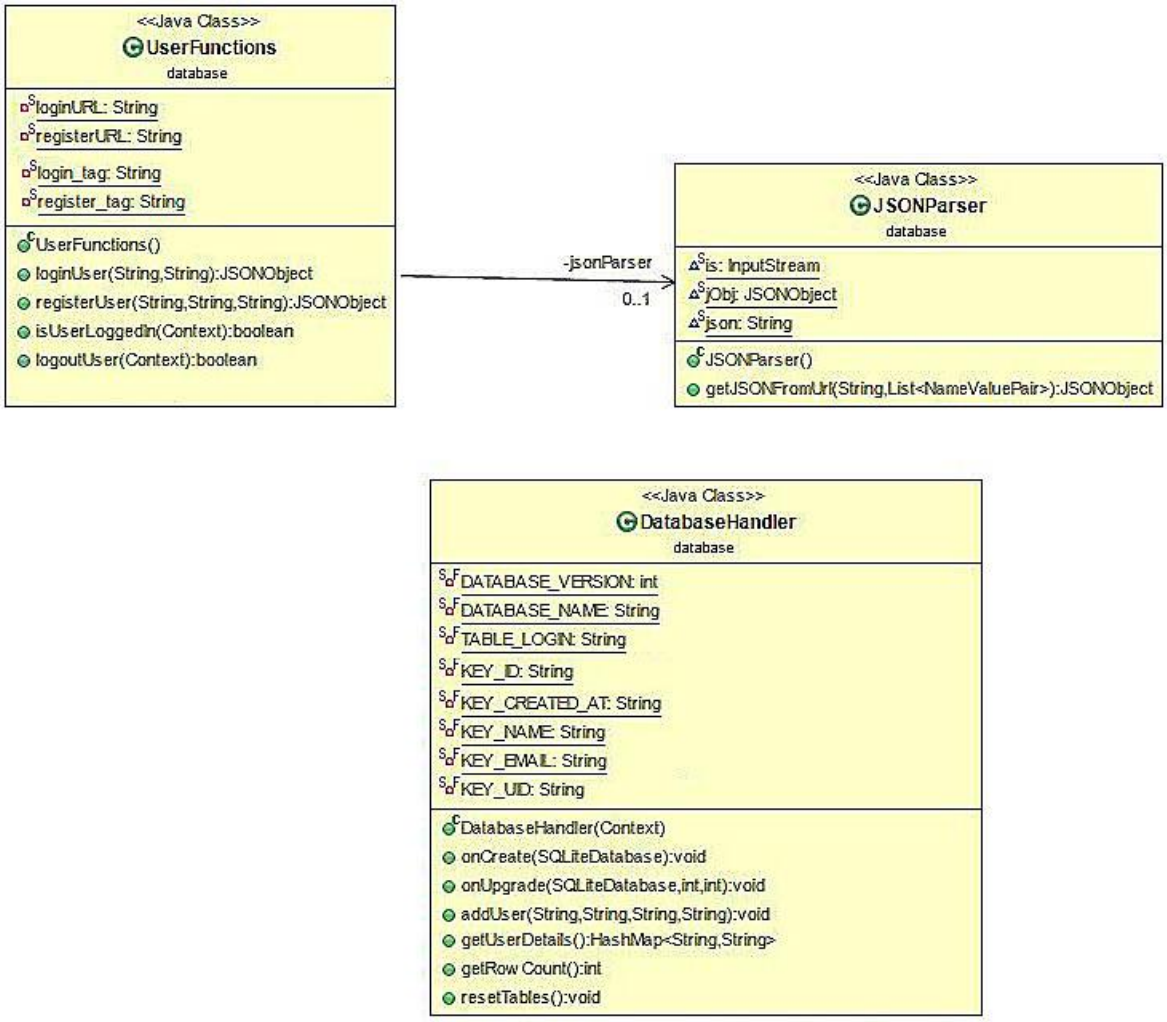


Figura 30 – Diagrama de classes da API (aplicação *Android*)

3.5. *GOOGLE MAPS*

O *Google Maps* é um serviço de pesquisa e visualização de mapas e imagens de satélite da Terra, gratuito e desenvolvido pela empresa Google, Figura 31.

O Google fornece uma API para aplicações WEB, denominada *Google Maps Javascript API v3*.

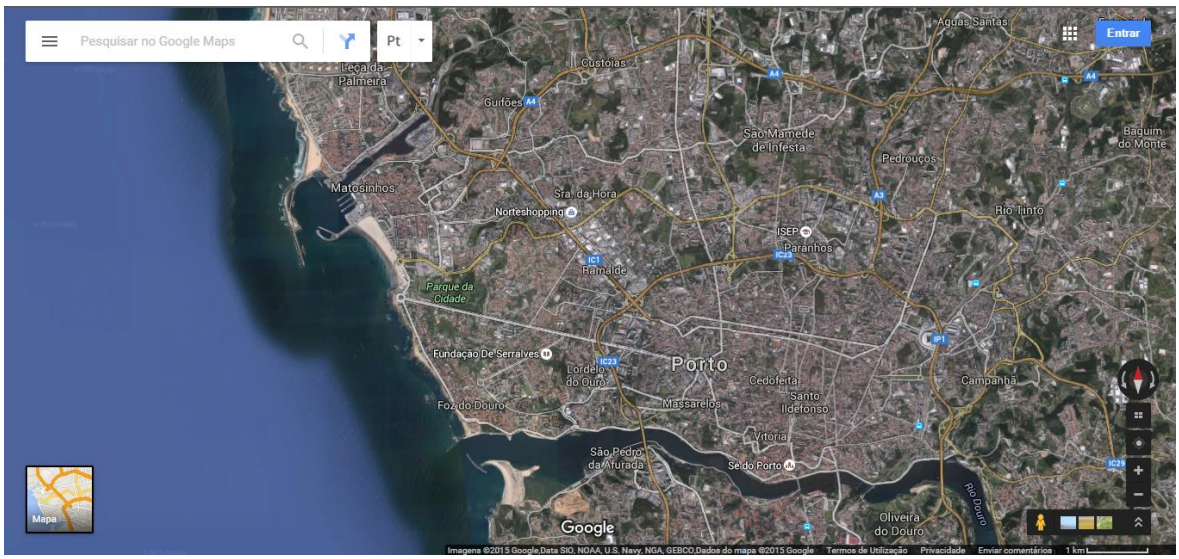


Figura 31 – Exemplo do Google Maps

O *Google Maps* pode ser implementado numa página WEB, realizando por exemplo, as seguintes operações [36]:

- Declarar a aplicação como HTML5 utilizando “<DOCTYPE html!>“;

```
// Declarar a aplicação como HTML5
<!DOCTYPE html>
```

- Criar um elemento “div” denominado "map" que alojar o mapa;

```
// Elemento div
<body>
    <div id="map"></div>
```

- Criar uma função *JavaScript* que cria um mapa na div “map”;

```
// Script
<script type="text/javascript">
var map;

function initMap(){
map = new google.maps.Map(document.
                    getElementById('map'),
    {// opções
    center: {lat: -34.397, lng: 150.644},
    zoom: 8
    });
}
    </script>
```

- Carregar o *GoogleMaps API JavaScript* usando uma "tag" de "script". O URL contido na "tag" "script" é o de um ficheiro *JavaScript* que carrega todos os símbolos e definições necessários para a utilização da API do *Google Maps*.
  O atributo "async" permite que o navegador WEB possa continuar a processar a página, enquanto a API é carregada. Quando esta estiver terminado o carregamento, será chamada a função especificada (*initMap*).

```
  // Script - GoogleMaps API JavaScript
  <script async defer
  src="https://maps.googleapis.com/maps/api/js?key=API_KEY&callback=initMap">
  </script>
```

O tipo de mapa apresentado ao utilizador, pode ser alterado nas opções de mapa aquando da sua criação. Constantes que definem o tipo de mapa apresentado:

- MAP_TYPE_HYBRID – Mapa com vista de satélite com uma sobreposição das vias principais;
- MAP_TYPE_NONE;
- MAP_TYPE_NORMAL – Mapa básico;
- MAP_TYPE_SATELLITE – Mapa com vista de satélite sem sobreposições;
- MAP_TYPE_TERRAIN – Mapa de terreno.

O tipo de mapa pode ser inicialmente definido nas opções de criação do mapa, bem como alterado dinamicamente, utilizando por exemplo:

```
  // Alteração para mapa de terreno
  map.setMapTypeId(google.maps.MapTypeId.TERRAIN);
```

4. DESENVOLVIMENTO

Este capítulo descreve o desenvolvimento da aplicação *Android* bem como da aplicação WEB para o servidor.

4.1. IDENTIFICAÇÃO DE REQUISITOS DE SISTEMA

Tendo como objetivo o desenvolvimento de um sistema AVL para dispositivos móveis com sistema operativo *Android*, é possível elaborar a lista dos requisitos principais para o equipamento:

- Sistema operativo *Android;*
- GPS interno;
- Acesso à internet;
- *Bluetooth.*

4.2. ARQUITETURA GERAL DO SISTEMA

O sistema idealizado será composto por quatro principais componentes, conforme ilustra a Figura 32:

- Aplicação *Android* de localização;

- Aplicação WEB de monitorização;
- Adaptador OBDII/*Bluetooth;*
- Dispositivos/Sensores *Bluetooth.*

O dispositivo *Android* com a aplicação de localização irá receber o sinal de GPS e transmite a sua localização geográfica através de redes móveis para um servidor WEB, que contém uma base de dados Mysql.

A aplicação *Android* permite ainda a obtenção de dados do veículo através de um adaptador OBDII/*Bluetooth* e de sensores que comuniquem por *Bluetooth* (até um máximo de sete dispositivos ou seis no caso da utilização simultânea com o adaptador OBDII/*Bluetooth*), enviando-os posteriormente para um servidor WEB conjuntamente com os dados de localização.

Figura 32 – Arquitetura geral do sistema

Quanto à aplicação *Android* esta será composta por:

- Um ecrã de *login* e registo no sistema;

- Dois ecrãs de visualização de dados;
- Um serviço executado em *background*, permitindo assim, utilizar outras funcionalidades do dispositivo móvel em simultâneo;
- Um ecrã de preferências para configuração de alguns parâmetros da aplicação.

A aplicação WEB de monitorização permitirá a um utilizador/gestor de frota visualizar num mapa interativo (*Google Maps*), a última localização de cada um dos utilizadores registados no sistema, consultar dados provenientes de cada veículo (OBDII) bem como dos sensores instalados. Fornecerá também ao gestor/utilizador um ecrã de login no sistema.

O servidor WEB possuirá uma aplicação, em linguagem de programação *PHP: Hypertext Preprocessor* (PHP), para interação com uma base de dados MYSQL, que armazenará todos os dados enviados pela aplicação *Android*.

O sistema dependerá assim das seguintes tecnologias:

- *Bluetooth*;
- *On-Board Diagnostic II* (OBDII);
- Sistema operativo *Android;*
- *Google Maps Javascript API;*
- Base de dados MYSQL;
- Aplicação PHP.

4.3. APLICAÇÃO *ANDROID*

A aplicação *Android* desenvolvida é responsável por gerir a comunicação com o adaptador OBDII/*Bluetooth*, possíveis sensores *Bluetooth* adicionais, aquisição da posição geográfica do dispositivo e transmitir estes dados para um servidor WEB. Esta aplicação é composta por um serviço *Android* e por quatro atividades ("Dashboard/OBDII", "GPS/Sensors", *Login* e Registo).

A Figura 33 ilustra a arquitetura geral da aplicação *Android* e o modo como o utilizador poderá interagir com a aplicação. A aplicação inicia-se através das atividades de login e registo onde o utilizador poderá autenticar-se e/ou registar no sistema. Completando uma destas etapas a atividade *Dashboard/OBDII* surgirá no ecrã. Posteriormente o utilizador

poderá abrir a atividade das preferências de modo a configurar alguns parâmetros da aplicação, iniciar/interromper o serviço *Android* de localização ou mudar para a atividade GPS/Sensores.

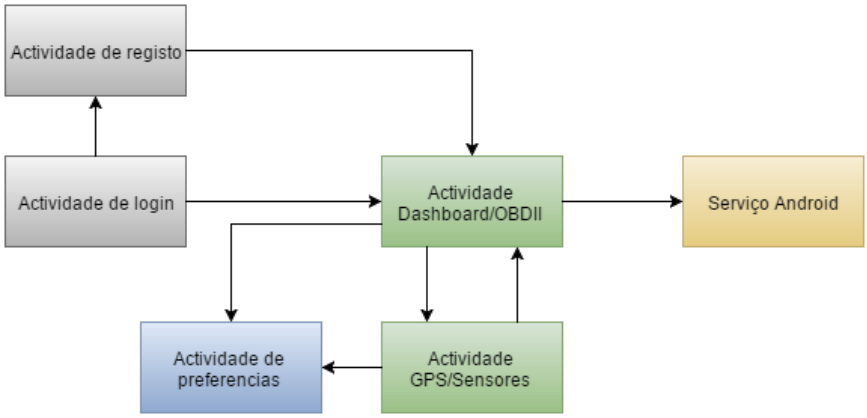


Figura 33 – Arquitetura geral da aplicação *Android*

4.3.1. ATIVIDADES *DASHBOARD/OBDII* E *GPS/SENSORS*

A atividade principal da aplicação é na realidade composta por dois *Fragments*, de modo a serem criados dois ecrãs distintos, Figura 34, selecionáveis pelo utilizador através de um simples *click* nas *tabs* da *actionbar* (1).

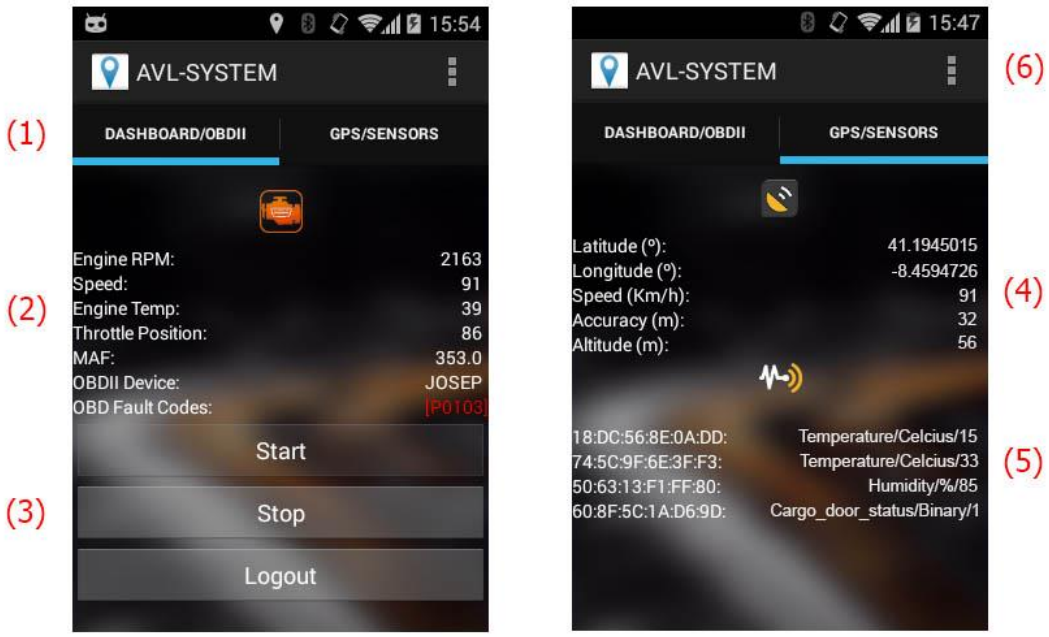


Figura 34 – Atividade "Dashboard/OBDII" (à esquerda) e atividade "GPS/sensors" (à direita)

Estas duas atividades são então compostas por:

- *Tabs* da *Actionbar* que permite selecionar qual a atividade visível ao utilizador (1);
- Campo de dados provenientes do veículo via OBDII/*Bluetooth* (2);
- Botões de controlo da aplicação (3);
- Dados GPS de localização (4);
- Dados dos sensores *Bluetooth* (5);
- Botão para acesso às definições da aplicação (6).

A. Preferências da aplicação

A aplicação desenvolvida permite ao utilizador modificar alguns dos parâmetros necessários ao seu funcionamento. A Figura 35 ilustra o ecrã de preferências mostrado ao utilizador. As opções disponíveis bem como a respetiva descrição estão enumeradas na Tabela 15.

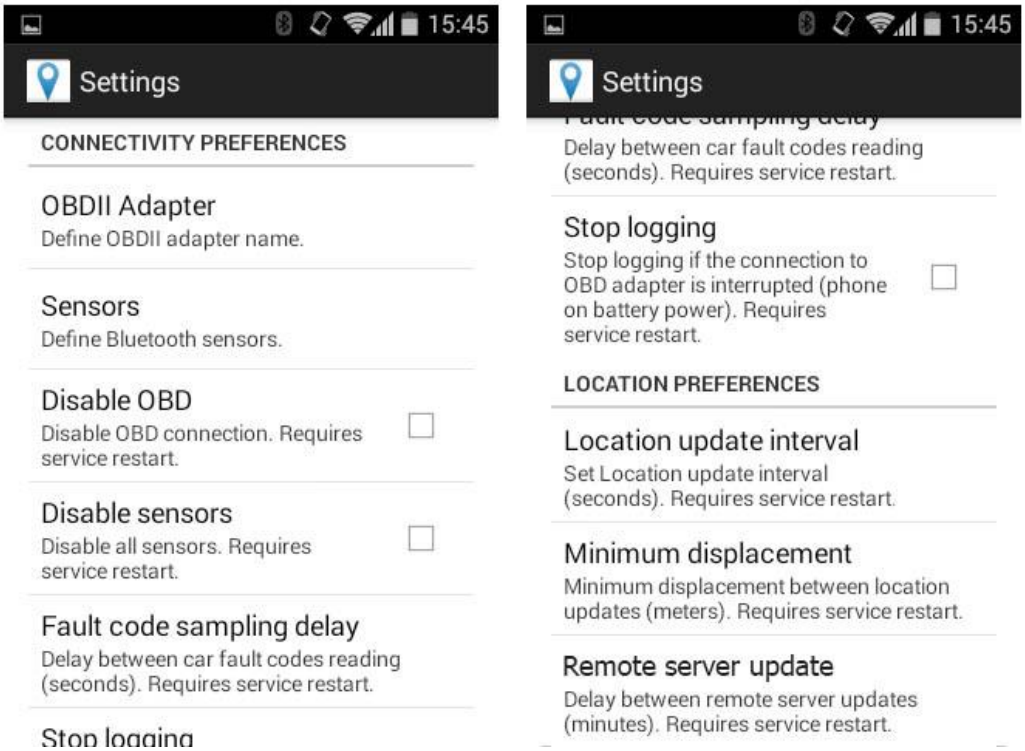


Figura 35 – Ecrã de preferencias da aplicação

A implementação do ecrã de preferências teve como orientação o guia da API *Android* [38] Esta inicia-se com a criação de um ficheiro XML que contém as preferências disponíveis para o utilizador. É necessário especificar qual o tipo de preferência para cada opção bem

como os seus atributos, tal como o valor pré-definido da opção. A chave (*key*) é outro atributo que é necessário especificar de modo às opções serem guardadas automaticamente (*sharedPreferences*).

Tabela 15 – Opções da aplicação Android desenvolvida

Opção	Tipo de preferência	Descrição
OBDII Adapter	Edittext	Nome do adaptador OBDII/*Bluetooth*
Sensors	MultiSelectList	Selecionar sensores *Bluetooth*
Disable OBD	Checkbox	Desativar ligação ao adaptador OBDII
Disable sensors	Checkbox	Desativar ligação a todos os sensores
Faultcode sampling delay	Edittext	Tempo entre leitura dos DTC's do veículo
Stop Logging	Checkbox	Desativar o registo de localizações na base de dados interna da aplicação e servidor (se desligado do adaptador OBDII e o telefone não estiver ligado via USB/AC)
Location update interval	Edittext	Tempo entre atualizações de localização
Minimum displacement	Edittext	Deslocação mínima do dispositivo em metros entre localizações
Remote server update	Edittext	Tempo entre envio de localizações para o servidor WEB

O código XML seguinte exemplifica a definição de uma *EditTextPreference* que permitirá ao utilizador definir o tempo entre atualizações de localização por parte do *LocationManager*:

```
   <PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/andr
oid">

   // EditTextPreference
   <EditTextPreference
      // Unique key
      android:key="update_interval"
      // Tipo de dados de entrada
      android:inputType="number"
      // Titulo da preferência
      android:title="
              @string/pref_title_update_interval"
      // Descrição
      android:summary="  ´
             @string/pref_summary_update_interval"
      // Valor Prédefinido
      android:defaultValue="
          @string/pref_default_update_interval" />
```

De seguida é criada uma classe *SettingsFragment* que estende a classe *PreferenceFragment.* Esta irá adicionar as preferências do ficheiro XML anteriormente definido para o mostrar ao utilizador.

Por fim uma *Activity* é criada e o seu conteúdo é substituído pelo conteúdo do fragmento.

```
  public class SettingsActivity extends Activity
{
      @Override
     protected void onCreate(Bundle
savedInstanceState) {

    super.onCreate(savedInstanceState);

   // Display the fragment as the main content.
     getFragmentManager().beginTransaction()
                 .replace(android.R.id.content, new
                  SettingsFragment())
                 .commit();
      }
  }
```

B. Atividade *Dashboard/OBDII*

Esta atividade permite ao utilizador iniciar/interromper um serviço, que será responsável por comunicar com o adaptador OBDII/*Bluetooth*, possíveis sensores *Bluetooth* adicionais, aquisição da posição geográfica do dispositivo e transmitir estes dados para um servidor web. É disponibilizado ao utilizador a informação recolhida do adaptador OBDII/*Bluetooth*, e a possibilidade de iniciar ou interromper o serviço.

A criação da atividade começa pela definição de uma classe denominada *StatusActivity* que estende os métodos da classe *Fragment*. São inicializadas as variáveis necessárias, botões e *intents* utilizados. São também definidas variáveis que armazenarão as preferências do utilizador referentes aos dispositivos *Bluetooth* a utilizar.

No método *onCreateView* da atividade (executado na inicialização da atividade) é realizado uma:

- Verificação do estado do serviço através do método *isServiceRunning* que retorna "true" ou "false" [40]. Este processo é efetuado para atualizar o estado dos botões "Start" e "Stop";
- Modificação do estado dos botões "Start" e "Stop";

- Leitura do nome do dispositivo OBDII das preferências da aplicação (*sharedPreferences*);
- Leitura dos endereços MAC dos dispositivos *Bluetooth* sensores das preferências da aplicação (*sharedPreferences*).

De seguida é definido um *event listener* para o botão de *logout* que reporta um *click*. Este botão é responsável pelo *logout* do sistema. Este processo é efetuado, em termos gerais, da seguinte forma (processo semelhante para o botão "close"):

- Chamada ao método *logoutUser* da classe *UserFunctions*, que irá apagar os dados de autenticação referentes ao utilizador da base de dados interna da aplicação;
- Verificação do estado do serviço através do método *isServiceRunning* que retorna "true" ou "false";
- O serviço é parado;
- Iniciada a aplicação de *login.*

Posteriormente é definido o *event listener* para o botão de *start* que reporta um *click*. Este botão é responsável pela inicialização do serviço.

De forma a inicializar o serviço é necessário realizar algumas operações e verificações iniciais, tais como:

- Obter o *BluetoothAdapter* para verificação da conectividade *Bluetooth*;
- Verificar se o dispositivo suporta a tecnologia *Bluetooth*;
- Verificar o estado do *Bluetooth* (o serviço só é iniciado caso este esteja ativo. É pedido ao utilizador a ativação o *Bluetooth*);
- Obter o *Location Manager* para verificação do estado do GPS e notificar o utilizador caso este esteja desligado (o serviço só é iniciado caso este esteja ativo).

Após a conclusão destas operações o serviço é iniciado. O serviço implementado envia para esta atividade, dados obtidos do adaptador OBDII/*Bluetooth*. De forma à atividade receber estes dados é criado um *Broadcast Receiver*.

No momento de receção dos dados é iniciado um *bundle*, habitualmente utilizado para a troca de dados entre várias atividades *Android*, de forma a obter todos os dados enviados pelo serviço, utilizando o método *intent.getExtras()*.

O serviço envia uma notificação (através de um *intent*), em que cada variável é descrita por uma "tag" de identificação. A Tabela 16 ilustra as "tags" recebidas pela atividade "Dashboard /OBDII" e a sua funcionalidade.

Desta forma é possível separar as informações enviadas pelo serviço para a atividade e atualizar, as respetivas caixas de texto (visíveis ao utilizador) com os valores obtidos.

Tabela 16 – "Tags" recebidas pela atividade "Dashboard /OBDII"

TAG	Funcionalidade
ENGINERPM	Rotações do motor do veículo (OBDII)
FAULTCODES	DTC's do veículo (OBDII)
VEHICLESPEED	Velocidade do veículo (OBDII)
ENGINECOOLANTTEMP	Temperatura do líquido de refrigeração do motor do veículo (OBDII)
ENGINEMAF	Sensor MAF (OBDII)
ENGINETHR	Posição do acelerador (OBDII)
BLUE	Notificação (caso o *Bluetooth* seja desligado)

C. Atividade GPS/Sensors

A atividade "GPS/Sensors" recebe os dados de localização GPS do serviço *Android*, bem como os dados provenientes dos sensores *Bluetooth*. Estes dados são apresentados ao utilizador como visível na Figura 34. A sua implementação começa com a criação da classe *LocationStatusActivity* que estende a classe *Fragment.*

No método *onCreateView* são definidos o *layout* da atividade, *textview's* e uma lista de *Strings*. A lista irá conter o endereço MAC de cada um dos sensores *Bluetooth* escolhidos pelo utilizador nas preferências da aplicação.

Esta lista irá permitir que os sensores sejam apresentados na atividade por uma determinada ordem (a mesma por que estão definidos nas preferências). Isto possibilita associar duas *textview* a cada sensor com base da sua posição na lista (uma para apresentação do nome do sensor e outra para o valor do sensor).

A Figura 36 ilustra este processo. Por exemplo: Para um sensor na posição “1" da lista, o nome do dispositivo estará apresentado na *textview10* e o seu valor na *textview11*.

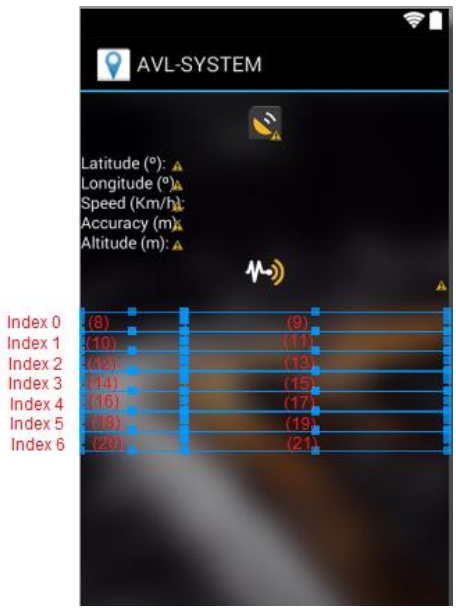


Figura 36 – Caixas de texto na actividade GPS/Sensors para apresentação de sensores

```
// Sensores escolhidos pelo utilizador

(...)
Set<String> sensorspref =
preferences.getStringSet("Devices", null);

   if(sensorspref!=null)
   {
   selected = sensorspref.toArray(new String[] {});
   list = new
ArrayList<String>(Arrays.asList(selected));
   (...)
```

Da mesma forma efetuada na aplicação *Dashboard/OBDII*, a troca de dados entre o serviço *Android* é realizada através de *BroadcastReceivers*, neste caso, dois *BroadcastReceivers* (um para os dados de localização GPS e outro para os sensores).

Tabela 17 – “Tags” utilizadas pelo *broadcastreceiver* associado aos sensores

TAG	Identificação
SENSOR	Mensagem do sensor
DEVICE	Endereço MAC do sensor
DEVICE_NAME	Nome do sensor

A Tabela 17 ilustra as *tags* de identificação relativas ao *broadcastreceiver* associado aos sensores. No caso do *broadcastreceiver* associado às informações de localização (GPS) são utilizadas as *tags* de identificação visíveis na Tabela 18.

Tabela 18 – "Tags" utilizadas pelo *broadcastreceiver* associado às informações de localização

TAG	Identificação
LAT	Latitude
LONG	Longitude
SPEED	Velocidade
ACCURACY	Acurácia
ALTITUDE	Altitude

Assim, primeiro são obtidos os dados enviados pelo serviço (método *onReceive* da classe *BroadcastReceiver*) e posteriormente é feita a atribuição destes valores às respetivas *textview*.

```
for(int i=0; i<list.size(); i++)
{
      if (list.get(i).equals(device)) {

      // Index 0
      if(i==0)
      {
      textView8.setText(device_name);
      textView9.setText(msg);
}
      // Index 1
      else if (i==1)
            {
             textView10.setText(device_name);
             textView11.setText(msg);
             }
(...) else if
```

4.3.2. SERVIÇO *ANDROID*

O serviço *Android* desenvolvido é responsável pelas seguintes operações, Figura 37:

- Aquisição da posição geográfica do dispositivo (GPS);
- Comunicar com o adaptador OBDII/*Bluetooth* e sensores *Bluetooth*;

- Armazenar os dados obtidos na base de dados interna da aplicação *Android*;
- Transmitir dados adquiridos para um servidor WEB;
- Notificar as atividades *Android* dos dados obtidos.

Como visto na secção 3.3.3, um serviço é um possível componente de uma aplicação *Android*, que pode executar operações em segundo plano (*background*). Uma atividade pode iniciar um serviço e ele continuará a sua execução em segundo plano mesmo que o utilizador aceda a outras aplicações.

Uma consideração importante é o facto de um serviço ser executado na *thread* principal do processo que o hospeda, ou seja, o serviço não cria a sua própria *thread*. Isso significa que para a realização de tarefas no serviço que envolvam o uso intensivo do CPU ou bloqueio de operações (como operações de rede), deve ser criada uma nova *thread* de modo a que a *thread* principal da aplicação permaneça dedicada às interações com o utilizador.

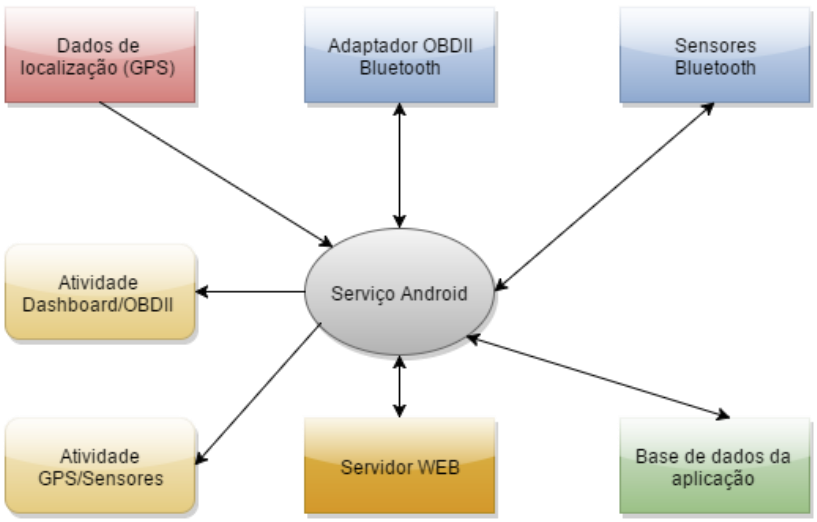


Figura 37 – Arquitectura do serviço Android

1. **Condições para gravação de dados na base de dados e posterior envio para o servidor WEB**

Na aplicação é dado ao utilizador a opção de a sua localização não ser gravada (e enviada para o servidor) se este não estiver na viatura, sem o utilizador ter de interromper o serviço de localização manualmente. A forma encontrada para fazer esta deteção sem recorrer a

sensores, é utilizar o estado da conexão com o dispositivo *Bluetooth* OBDII e o estado da alimentação do próprio telefone.

Tabela 19 – Possivel posição do utilizador

Estado da ligação OBDII	Alimentação do telefone (USB ou AC)	Possível posição do utilizador
Desligado	Desligado	Não se encontra no veículo
Desligado	Ligado	Não é possível determinar. Assume-se que se encontra no veículo.
Ligado	Desligado	Encontra-se no veículo
Ligado	Ligado	Encontra-se no veículo

A Tabela 19 ilustra os possíveis estados da alimentação do telefone e ligação OBDII. De acordo com estes estados é possível (com alguma margem de erro), determinar se o utilizador se encontra dentro ou fora do veículo (o adaptador *Bluetooth*/OBDII tem um alcance físico de cerca de 10 metros).

Desta forma, o registo da localização do utilizador não é gravado na base de dados interna da aplicação nem enviado para o servidor. Assim torna-se necessário fazer a verificação do estado da alimentação do telefone e da ligação com o adaptador OBDII/*Bluetooth*.

Para o serviço ser notificado de alterações no estado da alimentação do telefone, é utilizado um *BroadcastReceiver*.

2. Aquisição da posição geográfica do dispositivo;

O desenvolvimento deste serviço *Android* inicia-se pela criação de uma classe, *AndroidService*, que estende métodos da classe *Service* do *Android* e implementa um *LocationListener*.

O *LocationListener* é utilizado para receber notificações dos serviços de localização do *Android* no caso de a localização do dispositivo ser alterada segundo determinados parâmetros. Neste caso a distância percorrida pelo veículo e uma constante de tempo (definidas pelo utilizador nas preferências da aplicação), determinam a frequência de

atualização de notificações da posição GPS. No momento de alteração da localização do dispositivo é chamado automaticamente o método *onLocationChanged*.

Este método é responsável por receber os dados de localização do *LocationManager*, e realizar as seguintes operações*:*

- Notificar a atividade *GPS/Sensores* da posição atual do dispositivo;
- Ordenar a execução da *asynctask "GravaDBLocTask"*.

A execução da *asynctask* "*GravaDBLocTask"* depende da preferência definida pelo utilizador (*DISABLE_LOG*) e pelas condições atuais de registo (*logconditions*). Desta forma, se a opção "DISABLE_LOG" estiver ativa, a localização do utilizador só é guardada na base de dados caso as condições de registo o permitam ("logconditions==true").

3. Comunicação Bluetooth

A documentação incluída com o SDK *Android* fornece um exemplo da implementação de uma classe que permite a comunicação entre dois dispositivos *Bluetooth* utilizando a API *Bluetooth* [33].

Esta classe não permite no entanto, a comunicação com mais do que dois dispositivos, nem restabelecer uma ligação em caso de uma qualquer falha, pelo que foi necessária a sua modificação.

A classe "BluetoothConnectionManagement" implementada é composta por três principais classes, que implementam as suas funcionalidades através de *threads* (estendem a classe *Thread)*:

- "ConnectThread" – *Thread* executada durante a tentativa de ligação a um dispositivo. Tentará obter um *RFcomm socket* e posteriormente ligar ao dispositivo;
- "ConnectedThread" – *Thread* executada durante uma ligação com um dispositivo, manuseando todas as transmissões (de e para o dispositivo). Existirá uma "ConnectedThread" associada a cada dispositivo ligado;
- "ReConnectedThread" – *Thread* executada para efetuar a ligação a um dispositivo caso a anterior tentativa falhe.

O processo de ligação de um dispositivo é exemplificado na Figura 38.

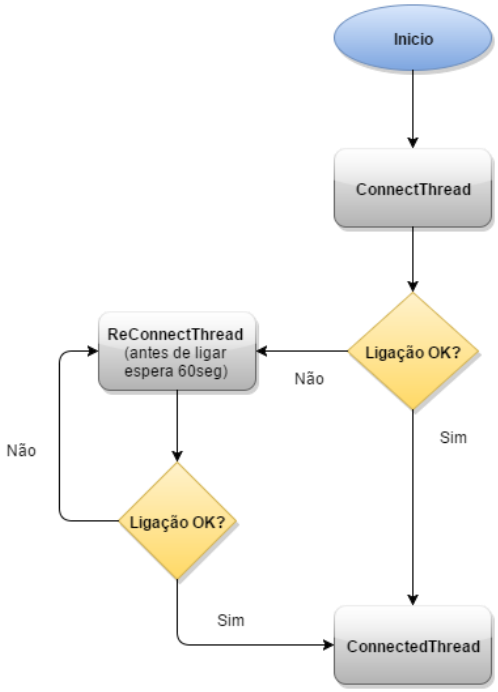


Figura 38 – Processo de ligação da aplicação a um dispositivo *Bluetooth*

Para comunicar e gerir a ligação *Bluetooth* com múltiplos dispositivos *Bluetooth* torna-se necessário determinar quais os dispositivos ligados, dispositivos ainda em estado de ligação e qual a *ConnectedThread* de ligação associada a cada dispositivo num dado instante.

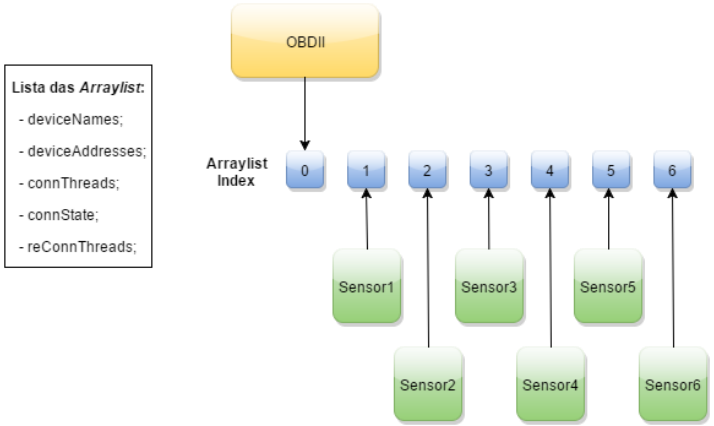


Figura 39 – Atribuição de posição dos Arraylist aos dispositivos

A forma encontrada de viabilizar este processo é criar cinco *ArrayList* que contenham os dados associados a cada um dispositivo (nome do dispositivo, endereço MAC, *ConnectedThread*, *ReConnThread* e estado da ligação*)* a cada instante. O adaptador

OBDII/*Bluetooth* é sempre atribuído à posição 0 e os sensores às posições subsequentes, Figura 39.

De modo a iniciar o processo de ligação com os dispositivos *Bluetooth* externos, é chamado o método "connect" da classe "BluetoothConnectionManagement", para cada um dos dispositivos (este método requer o endereço do dispositivo *Bluetooth* e a posição dos *Arraylist* a que ficará associado como parâmetros).

No caso do adaptador OBDII/*Bluetooth*:

```
if(DISABLE_OBD==false)
(...)
  if(mmDevice!=null)

   // LIGAR OBD, posição 0
   FD.connect(mmDevice, 0);
```

Para os sensores o processo é semelhante, sendo feita a ligação aos dispositivos selecionados nas preferências da aplicação. Posteriormente é efetuada a ligação a cada um dos dispositivos (sensores). Após a chamada ao método "connect" da classe "BluetoothConnectionManagement", é criada uma "ConnectThread" que irá tentar obter um *BluetoothSocket* para o dispositivo *Bluetooth* e ligar ao dispositivo.

Caso esta tentativa tenha sucesso, os *Arraylist* correspondentes ao nome e endereço do dispositivo são preenchidos e o método "connected" da classe "BluetoothConnectionManagement" inicia a "ConnectedThread". A "ConnectedThread" irá adquirir o "inputStream" e "outputStream" associado ao "Bluetoothsocket" anteriormente criado.

Enquanto o dispositivo estiver ligado, esta *thread* estará à espera de um caracter terminador no "inputStream". Este caracter sinalizará o fim de uma mensagem enviada pelos dispositivos *Bluetooth*. No caso do adaptador OBDII/*Bluetooth* (baseado no microcontrolador ELM327), este envia o caracter ">" no fim de cada mensagem. Assim e de modo a simplificar o processo de sinalização de fim de mensagem, este caracter também será utilizado para as mensagens provenientes dos sensores *Bluetooth*.

No que diz respeito a erros de ligação, é possível distinguir dois tipos: perda de ligação e falha de ligação.

Se existir uma tentativa de ligação com um dispositivo e esta não tiver sucesso, o método "connectionFailed" da classe "BluetoothConnectionManagement" é chamado, que irá alterar o estado da ligação no *Arraylist* "connState". De seguida é chamado o método "Reconnect" da classe "BluetoothConnectionManagement", que irá iniciar uma nova "ReconnectThread" e notificar o serviço na operação.

No caso de uma perda de ligação (ligação já existente foi interrompida), o método "connectionLost" da classe "BluetoothConnectionManagement" é chamado. Este método modifica os *ArrayList* na correspondente posição do dispositivo e chama o método "Reconnect" da classe "BluetoothConnectionManagement" de modo a iniciar uma "ReconnectThread".

No que diz respeito à "ReconnectThread", esta funciona de modo semelhante à "ConnectThread", com a exceção de introduzir uma pausa de 60 segundos antes da próxima tentativa de ligação. Esta pausa é necessária para evitar falhas de ligação com o adaptador OBDII/*Bluetooth*.

Para enviar uma mensagem para um dispositivo é utilizado o método "write" da classe "BluetoothConnectionManagement". Este método necessita de determinar qual a "ConnectedThread" associada ao dispositivo destinatário da mensagem. Para isso bastará consultar o *Arraylist* "connThread" na posição indicada.

4. Troca de mensagens com o adaptador OBDII/Bluetooth e sensores

Para a troca de dados entre as *threads* de ligação e o serviço é utilizado um *Handler*, que é um manipulador que permite enviar e recuperar mensagens e objetos executáveis associados com uma *thread* de execução. Cada instância de um *handler* está associado a uma única *thread* e à sua fila de mensagens [39].

Assim, no construtor da classe "BluetoothConnectionManagement" é definido um *handler* associado ao adaptador OBDII/*Bluetooth*. Este é definido no serviço *Android* na instanciação da classe "BluetoothConnectionManagement".

Posteriormente é possível definir ações dependendo da mensagem recebida.

No caso de uma mensagem recebida, é enviado ao serviço a posição do dispositivo nos *Arraylist* e a mensagem enviada pelo dispositivo:

```
  private final Handler mHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
    switch (msg.what) {

    (...)
  case MESSAGE_READ:

  Bundle bundle = msg.getData();
  bundle.setClassLoader(Thread.currentThread().getCon
textClassLoader());
  String msg1=bundle.getString("msg");

  // Posição do dispositivo nos ArrayList
  int posicao = bundle.getInt("deviceposition");

  // Chama método mensagemRecebida
  mensagemRecebida(msg1,posicao);
              break;
```

A Tabela 20 indica quais as mensagens recebidas pelo serviço *Android*.

Tabela 20 – Mensagens recebidas pelo serviço Android

Mensagens recebidas no serviço *Android*	**Parâmetros**	**Descrição**
STATE_NONE	Posição	Indica falha ou perda de ligação a um dispositivo Caso este dispositivo seja o adaptador OBDII/*Bluetooth* faz *reset* a todas as suas variáveis de inicialização
MESSAGE_READ	Posição Mensagem	Mensagem recebida. Chama o método "mensagemRecebida"
MESSAGE_DEVICE_NAME	Posição Nome do dispositivo	Nome do dispositivo ligado Envia mensagem inicial para o dispositivo
MESSAGE_TOAST	Nome do dispositivo	Indica que um dispositivo perdeu a ligação
MESSAGE_TOAST2	Nome do dispositivo	Indica que a falha de ligação a um dispositivo

A troca de mensagens com os dispositivos *Bluetooth* utilizados é efetuada com recurso a um sistema de pedido-resposta. O dispositivo *Android* envia um comando para os dispositivos e

espera uma resposta, de modo a posteriormente enviar um novo comando. Para implementar este sistema é necessário verificar antes de mais, qual o dispositivo que enviou a mensagem. No método "mensagemRecebida" da classe "AndroidService" é então verificado qual a posição nos *arraylist* do dispositivo que enviou a mensagem. Como o adaptador OBDII/*Bluetooth* se encontrará sempre na posição 0, este é ponto de controlo.

```
   private void mensagemRecebida(String msg, int
position) {

   if(mBluetoothConnectionManagement!=null)
   {
   // MSG DE ALGUM SENSOR - envia "X" para sensor
   if(0 != position)
   (...)
```

A Figura 40 ilustra o formato de mensagem estabelecido para a comunicação dos sensores com o dispositivo *Android*. O primeiro campo indica qual o tipo de sensor, ou seja, se é um sensor de temperatura, de humidade, binário, entre outros. O terceiro campo assinala o valor obtido pelo sensor, enquanto o segundo campo indica quais as unidades do valor medido. Os campos são separados pelo caracter ";" de forma a ser possível diferenciá-los.

Figura 40 – Formato de mensagem enviada pelos sensores *Bluetooth*

Este formato permite que diferentes tipos de sensor possam interagir com a aplicação *Android*, dado que podem ser diferenciados e identificados facilmente.

Para solicitar o envio de dados pelos sensores a aplicação *Android* necessita apenas de enviar um caracter para o sensor, neste caso "X", indicando assim que a aplicação se encontra à espera do envio de uma nova leitura.
No caso do adaptador OBDII/*Bluetooth* o processo de troca de mensagens implementado é ilustrado na Figura 41.

A comunicação com o adaptador OBDII/*Bluetooth* é iniciada com o envio do comando "ATZ", que faz o *reset* do dispositivo. A Tabela 21 ilustra os comandos enviados para inicialização do dispositivo, resposta esperada e descrição do comando.

Após o envio de um comando, uma resposta do adaptador é recebida.

Assim para a inicialização do adaptador OBDII/*Bluetooth* são criadas variáveis de sinalização, bem como um *String Array* com os comandos de inicialização.

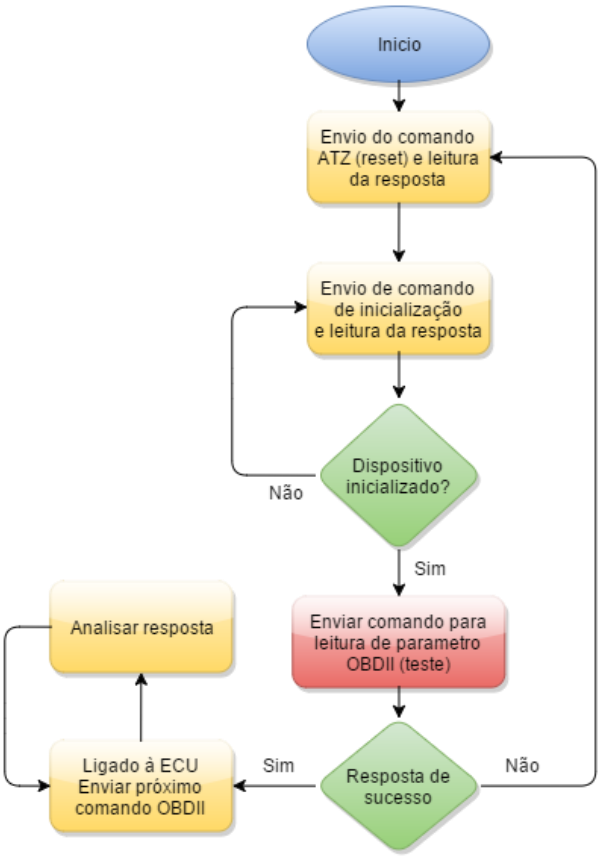


Figura 41 – Processo de comunicação com o dispositivo ELM327

Tabela 21 – Interpretação das respostas do ELM327 na fase de inicialização

Comando	Resposta esperada	Descrição
ATZ	ELMxxxx	Versão do dispositivo
ATL0	ATL0OK	Desativar *Linefeed*
ATH0	ATH0OK	Desativar cabeçalhos adicionais na resposta do ELM327
ATSPA0	ATSPA0OK	Escolher protocolo OBDII automaticamente
ATDP	ATDP + "protocolo"	Nome do protocolo
ATE0	ATE0OK	Desativar *echo* do comando enviado na resposta

Neste caso é necessário sinalizar três diferentes estados de ligação com o ELM327, pela seguinte ordem:

1. Não inicializado
2. Inicializado mas não ligado à ECU do veículo
3. Ligado à ECU do veículo

Desta forma é possível separar as mensagens recebidas por grupos (utilizando variáveis de sinalização), sem ser necessário analisar se mensagem é resposta por exemplo, a um comando que só é enviado se existir uma ligação com a ECU do veículo. Se for obtida a resposta esperada a todos os comandos de inicialização, o dispositivo é considerado inicializado e é enviado um comando de teste para verificar a ligação à ECU do veículo. No caso de a inicialização falhar o processo é reiniciado com o envio do comando "ATZ".

Após o envio do comando de teste que requisita a temperatura do líquido de refrigeração do motor do veículo (0105), o dispositivo no caso de estar ligado à ECU do veículo responderá com a mensagem "4105xx", o que permite concluir que o dispositivo está de facto ligado à ECU. Nesse caso a variável "ligadoECU" é colocada "true" e é enviado novamente o mesmo comando ao ELM327. Como o ELM327 já se encontra ligado à ECU, a análise da resposta será feita pelo segmento de código que analisa apenas respostas a comandos OBDII.

```
  if (inicializadoELM && !ligadoECU)
  {
    if (msg.contains("4105")) { // INIT OK

        // Ligado à ECU
        ligadoECU=true;
        if(mBluetoothConnectionManagement != null)

        // Envia próximo commando OBD
  AndroidService.this.sendMessage(ENGINE_COOLANT_TEMP
,FD.deviceposition(mmDevice));

   // Verificação de DTC's
   handlerFcodes = new Handler();
    handlerFcodes.postDelayed(runnableFaultCodes,
    FAULTCODEDELAY*1000);
     }

    else if (msg.contains("UNABLE"))
  {
  // Reinicialização
```

Neste segmento é também criado um *handler* que agendará a execução de um *runnable* para o tempo definido pelo utilizador nas opções na aplicação. Este é utilizado para sinalizar que

é altura de enviar um comando ao ELM327 para verificar os DTC's presentes na ECU do veículo. O *runnable*, no seu método *run()* fará apenas a mudança de estado de uma variável de sinalização utilizado posteriormente para verificação de qual comando OBDII será enviado num dado momento.

```
  private Runnable runnableFaultCodes = new
Runnable() {
              @Override
              public void run() {
                    FAULTCODESTIME=true;
              }};
```

Após o ELM327 estar ligado à ECU do veículo, é iniciado o processo de envio de comandos OBDII. Este processo é ilustrado na Figura 42 de uma forma simplificada.

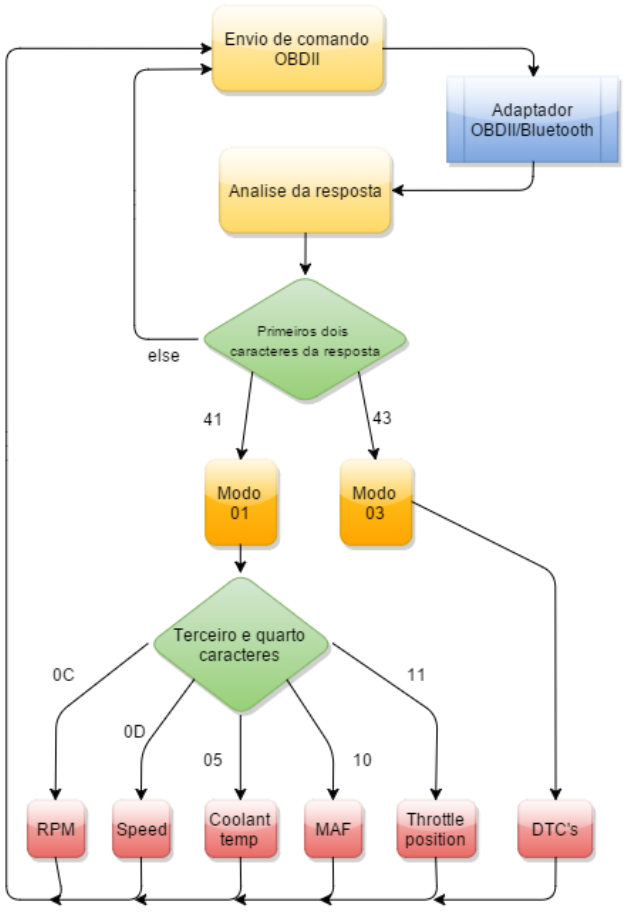


Figura 42 – Troca de comandos OBDII com ELM327 e analise da resposta

O processo pode ser classificado como um *loop* controlado. O envio de uma mensagem inicia o *loop*, dado que após a leitura de uma mensagem, o serviço envia o próximo comando OBDII da lista.

A Tabela 22 ilustra os comandos OBDII enviados para o ELM327. A escolha destes comandos está relacionada com o facto de serem parâmetros que, para além de indicarem o estado atual de funcionamento do veículo, estão disponíveis na grande maioria dos veículos compatíveis com OBDII, assegurando assim a compatibilidade da aplicação.

Tabela 22 – Comandos OBDII enviados para o ELM327

Comando OBDII enviado	Descrição
0105	Temperatura do líquido de refrigeração do motor
010C	Rotações do motor
010D	Velocidade do veículo
0110	Fluxo de massa de ar (MAF)
0111	Posição do acelerador

A resposta a um pedido no modo "01" é interpretada conforme o indicado na Tabela 6. No caso da resposta a um pedido no modo" 01" (41), são lidos os primeiros dois caracteres da resposta (primeiro *byte*), conforme ilustra a Figura 42.

Posteriormente são lidos o terceiro e quarto caracteres (segundo *byte*), de forma a determinar a que parâmetro OBDII (PID) a resposta se refere. Dependendo do valor deste, é aplicada a fórmula de conversão correspondente, Tabela 6, e posteriormente notificada a atividade "*Dashboard*/OBDII". Neste processo de leitura da resposta do ELM327 é também verificado o estado da variável de sinalização, que indica que deve ser enviado um comando OBDII para leitura dos DTC gravados na ECU.

No caso da resposta a um pedido no modo 03 (43), o procedimento de descodificação da mensagem é substancialmente diferente. Para obter os DTC's do veículo é necessário ter em consideração os seguintes factos:

- Primeiros dois carateres da resposta são sempre "43" (primeiro *byte*);
- Os carateres seguintes são os DTC's do veículo;
- Os carateres "43" são reintroduzidos na resposta a cada três DTC's (*multiple frames*).

O formato de uma possível resposta do ELM327 a um pedido no modo 03, é ilustrado na Figura 43, bem como o algoritmo implementado para descodificação da resposta. Este algoritmo funciona sem ser necessário enviar ao ELM327 um comando OBDII (modo "01"

e parâmetro “01”), o que permitiria saber antecipadamente qual o número de erros presentes na ECU do veículo.

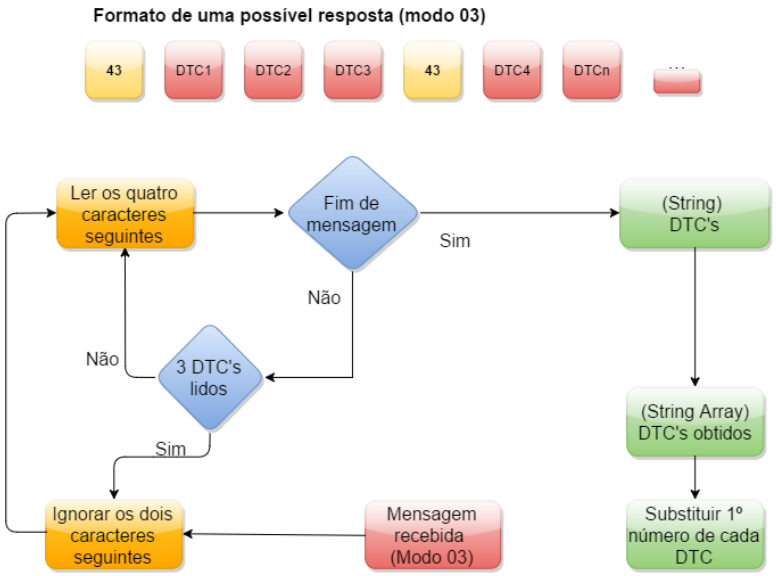


Figura 43 – Algoritmo implementado para interpretação da resposta do ELM327 a um pedido no modo “03”.

Caso exista mais do que um DTC, estes são separados efetuando uma conversão de *String* para *String Array [41]*, em que cada DTC ficará numa posição do *String Array*. Posteriormente é necessária a substituição do 1º caracter de cada um dos DTC, de acordo com o indicado na Tabela 7. Por fim notifica-se a atividade *Dashboard*/OBDII dos DTC presentes no veículo e envia-se um novo comando OBDII. O serviço *Android* continuará a enviar ao ELM327 comandos OBDII no modo “01” até que a variável “FAULTCODESTIME” seja “true”, momento em que o processo de leitura dos DTC’s será reiniciado novamente.

```
   // Enviar notificação para a atividade
      OBDII/Dashboard
   Intent intent = new Intent(NOTIFICATION);
   intent.putExtra(FAULTCODES, faultcodes);
   sendBroadcast(intent);

   // Agendar nova verificação de DTC's
   handlerFcodes.postDelayed(runnableFaultCodes,
FAULTCODEDELAY*1000*60);

   // Variável de sinalização -> false
   FAULTCODESTIME=false;

   // Envia novo comando OBDII da lista
   sendOBDCommand();
```

Os comandos OBDII são enviados com recurso ao método "sendOBDCommand" na classe "AndroidService":

```
   private void sendOBDCommand() {
   if (indexcomandoOBD > comandosOBD.length - 1) {

         indexcomandoOBD = 0;
         String send = comandosOBD[indexcomandoOBD];

         if(mBluetoothConnectionManagement != null)
         {
         // Enviar próximo comando
   AndroidService.this.sendMessage(send,
mBluetoothConnectionManagement.deviceposition(mmDevice
));
         indexcomandoOBD++;
         }

   } else {
         // Enviar próximo comando
         String send = comandosOBD[indexcomandoOBD];
         if (FD != null)

   AndroidService.this.sendMessage(send,
   mBluetoothConnectionManagement.deviceposition(mmDev
ice));
   indexcomandoOBD++;
                     }
```

5. Gravação de dados da base de dados e envio para o servidor WEB

A gravação de dados na base de dados da aplicação *Android*, tem como objetivo armazenar os dados temporariamente, até que seja possível o envio dos mesmos para o servidor WEB. O modelo físico da base de dados SQLite criada para o efeito é ilustrado na Figura 44. A base de dados é composta por cinco tabelas:

- "Login" – Dados de login do utilizador;
- "Locations" – Dados de localização (GPS);
- "Obddata" – Dados recolhidos do veículo (OBDII);
- "Sensordata" – Dados dos sensores Bluetooth;
- "Faultcodes" – DTC's do veículo.

De modo a efetuar o armazenamento de dados na base de dados e posterior envio para o servidor WEB, torna-se necessário determinar quais os registos a enviar e quais já foram enviados.

Este processo é efetuado pela alteração do campo "sent" da tabela "locations".

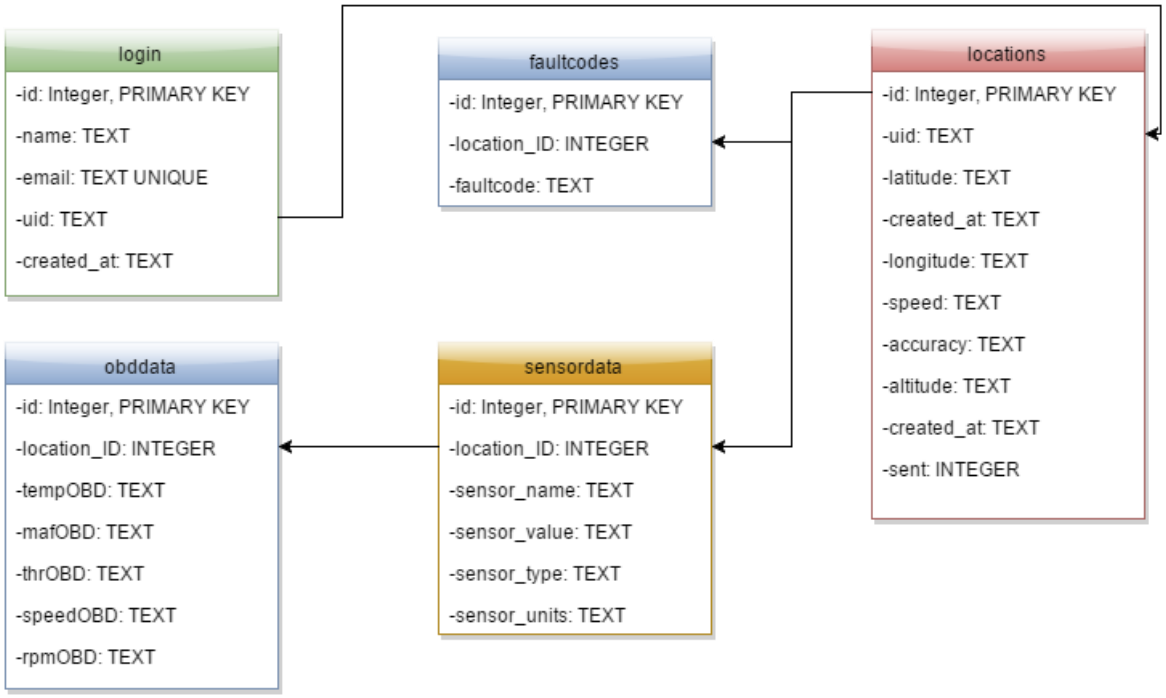


Figura 44 – Modelo físico da base de dados SQLite

Para efetuar a gravação dos dados na base de dados da aplicação e para o envio dos dados para o servidor WEB, foram criadas duas *asynctask*, respetivamente "GravaBDLocTask" e "EnviaLocTask" executadas em *background*. Tendo duas *asynctask*, o acesso simultâneo das mesmas à base de dados da aplicação é uma possibilidade.

De modo a prevenir este facto, foi implementado um sistema de semáforo com recurso às funcionalidades da API do *Android* [42]. O método "semaphore.acquire()" é bloqueante, ou seja, bloqueia a execução de uma *thread* até haver um "semaphore.release()" por parte de uma outra *thread*.

No caso da *asynctask* responsável por gravar os dados na base de dados da aplicação, após a aquisição da permissão de execução, o método "addLocationDB" da classe "DatabaseHandler" é chamado (que tem como parâmetros o valor atual de cada um dos dados recolhidos pelo sistema) e posteriormente o semáforo é libertado após a escrita dos dados.

Este método "addLocationDB" da classe "DatabaseHandler" recebe também duas "flags" que indicam se existem sensores ligados no momento e/ou um adaptador OBDII/Bluetooth. Deste modo determina-se quais as tabelas da base de dados a escrever.

A implementação do método "addLocationDB" inicia-se por abrir a base de dados em modo de escrita/leitura. De seguida os dados de localização GPS são colocados num *contentvalues* para posterior inserção na tabela "locations" da base de dados. A escrita dos dados da tabela é efetuada com o método "*insert"* da classe "SQLiteDatabase*"* que retorna o "id" do registo. O "id" do registo da tabela "locations" será atribuído aos campos "location_ID" das restantes tabelas, permitindo assim associar uma localização com os restantes dados.

Posteriormente os DTC's são inseridos na base de dados na tabela "faultcodes", os dados OBDII na tabela "obddata" e dados dos sensores na tabela "sensordata".

Para o envio dos dados para o servidor WEB surge a necessidade de temporizar este processo. No método "onCreate" do serviço é criado um *handler* de modo a iniciar um *runnable,* que irá executar a *asynctask* "EnviaLocTask" que inicia o processo de envio dos dados para o servidor, consoante o estado da ligação à *Internet* do telefone. O *runnable* é executado em intervalos de tempo definidos pelo utilizador nas preferências da aplicação. O processo inicia-se pela aquisição de permissão do semáforo para execução. Adquirindo a permissão do semáforo é chamado o método "getLocationsFromDB" da classe "DatabaseHandler", que retornará um *JSONArray*.

Este método "getLocationsFromDB" começa por solicitar à base de dados, todos os registos que não tenham sido marcados como enviados para o servidor (campo "sent"=0 na tabela *locations*). A leitura dos dados da base de dados SQLite no Android é efectuada com recurso a cursores (SQLiteCursor) [43].

Após a leitura de um registo da tabela "locations" é verificado se a tabela "faultcodes" possui algum registo com o campo "location_ID" igual ao "id" da tabela "location". Em caso afirmativo esses registos são anexados ao objeto JSON. O mesmo procedimento é efetuado na tabela "obdata" e "sensordata". Por fim o cursor relativo à tabela "locations*"* é fechado e o *JSONArray* "elements" é retornado.

Depois da obtenção do *JSONArray* com os dados da base de dados a serem enviados para o servidor, é chamado o método "addFULLLocation" da classe "userfunctions", com o *JSONArray* como parâmetro.

O método "addFULLLocation" da classe "userfunctions" começa por criar uma lista do tipo *NameValuePair* que armazenará uma "TAG" de identificação dos dados (para o servidor) a enviar e o *JSONArray* obtido anteriormente.

O envio dos dados fica a cargo do método "getJSONFromUrl" que retornará uma resposta em formato JSON (objeto) com o resultado da operação:

```
   public JSONObject addFULLLocation(JSONArray
elements) {

   // Criar lista de parametros
   List<NameValuePair> params = new
                          ArrayList<NameValuePair>();

   // adicionar TAG de identificação
   params.add(new BasicNameValuePair("tag",
                                addFULLlocation_tag));
   // adicionar JSONArray
   params.add(new BasicNameValuePair("json",
                                elements.toString()));

    // Envia e obtém a resposta do servidor
       JSON Object
    JSONObject json =
jsonParser.getJSONFromUrl(addLocationURL, params);

   return json;
       }
```

Após a obtenção da resposta é necessário verificar se o campo "success" dessa mesma resposta contém o valor um. Este valor é indicativo do sucesso da operação. Por fim é necessário marcar os registos na base de dados como enviados (campo "sent"=1 na tabela "locations"), recorrendo ao método "markLocSent" da classe "DatabaseHandler*"* e libertar o semáforo para as próximas operações na base de dados.

O método "markLocSent" da classe "DatabaseHandler*"* atualiza o campo "sent" de todos os registos da tabela "locations", dado que uma tentativa de envio dos dados da base de dados envia também todos os registos para o servidor.

4.3.3. ATIVIDADE – LOGIN

A atividade de *login* é responsável pela autenticação do utilizador no sistema. O *layout* desta atividade, bem como de toda a aplicação, foi desenhado com recurso ao editor gráfico de layouts do ambiente de desenvolvimento *Eclipse*.

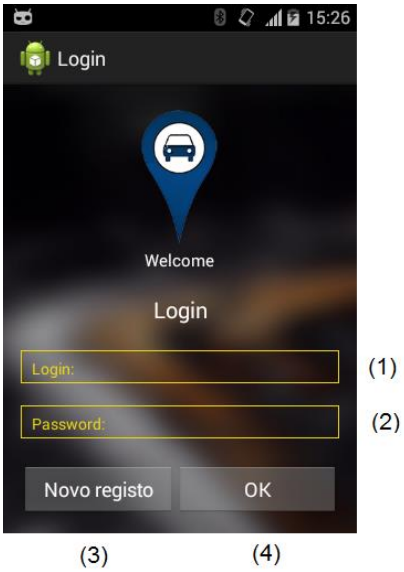


Figura 45 - Actividade de login

O *layout* das aplicações *Android* é definido através de ficheiros XML e pode ser editado tanto diretamente em linguagem XML como graficamente. A atividade de login é assim composta por duas caixas de texto editáveis, "email" (1) e "password" (2) e por dois botões, "Novo registo" (3) e "OK" (4).

A Figura 45 ilustra a atividade de *login* implementada. Quanto ao seu funcionamento, o botão "OK" efetua o login no sistema com os dados inseridos pelo utilizador e redireciona-o para a atividade principal da aplicação (*Dashboard*/OBDII). O botão "Novo registo" por sua vez redireciona o utilizador para a atividade de registo no sistema.

A Figura 46 ilustra o fluxo de dados nesta atividade, podendo ser resumido da seguinte forma:

1. O utilizador introduz os seus dados (*email* e *password*) e pressiona o botão "OK";
2. É iniciada a *asynctask* "LoadLoginTask" (uma tarefa *Android* que é executada em *background* assincronamente, dado que é obrigatório que se execute as operações de

rede numa *thread* independente da UI [37]) que chama o método *loginuser*() da classe *Userfunctions*, com o email e password como parâmetros;

3. Este método efetua a concatenação destes parâmetros e chama o método *getJSONFromUrl()* da classe *JsonParser*, que envia estes dados para o endereço web de destino;

4. O servidor envia a resposta ao pedido, em formato JSON para a atividade de *login.*

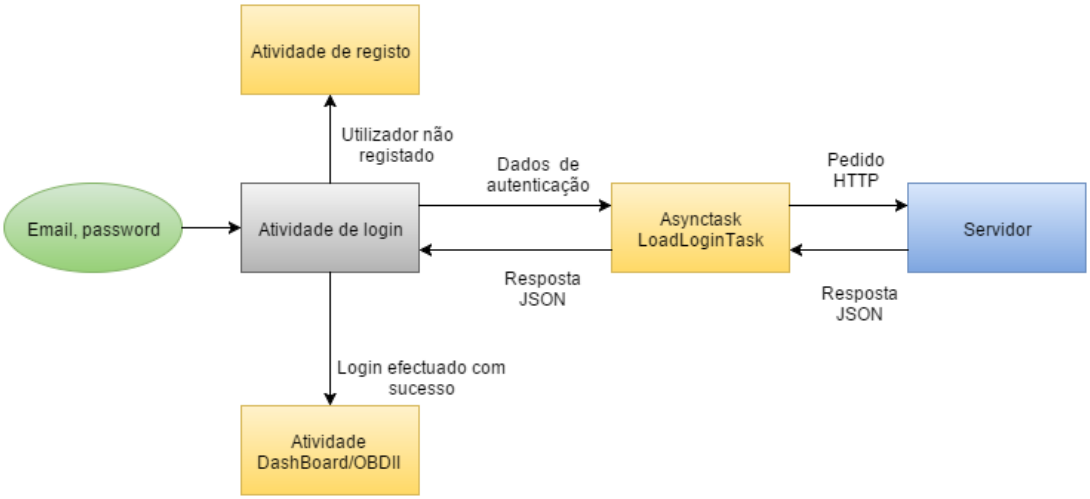


Figura 46 – Fluxo de dados da actividade de login

A implementação da atividade de *login* inicia-se pela criação da classe *LoginActivity* que estende a classe *Activity*, contida em *LoginActivity*,*java* que irá implementar os métodos necessários.

Depois é criado um *event listener* para o botão de registo que implementa o método *onClick*, cujo código é executado no momento de um *click* no botão. No momento de *click* no botão é criado um *Intent*, que funcionará como um *link* para a atividade de registo. Esta é iniciada através no método *startActivity()*, sendo que a atividade anterior é encerrada através do método *finish()*.

Para o botão de *login* é também criado um *event listener* que implementa o método "*onClick"*. Caso este botão seja pressionado, o texto introduzido nas caixas de texto pelo utilizador é copiado para as respetivas variáveis, *email* e *password*, e caso possuam texto é executada uma *asynctask*, "*LoadLoginTask"*, que irá fazer a comunicação com o servidor.

Serão utilizados três métodos da *asynctask*: "*onPreExecute*," "*doInBackground"* e "*onPostExecute*", e são executados pelo sistema por esta mesma ordem.

O método "*onPreExecute"* é executado na mesma *thread* que a UI e apresenta uma barra de progressão enquanto a autenticação estiver em curso, de forma ao utilizador ter de aguardar pelo fim de execução da *asyncTask*. É executado antes do método "*doInBackground"*.

O método "*doInBackground"* é o responsável por realizar a operação de rede. Executa o método "*loginUser"* contido na classe "*userFunction"* e retorna a resposta (objeto JSON) do servidor. O método "*loginUser"* recebe como parâmetros o *email* e a *password* do utilizador, e é criada uma lista de parâmetros que serão enviados ao servidor.

```
   public JSONObject loginUser(String email, String
password){

   // Lista de Parametros
List<NameValuePair> params = new
ArrayList<NameValuePair>();

params.add(new BasicNameValuePair("tag", login_tag));
params.add(new BasicNameValuePair("email", email));
params.add(new BasicNameValuePair("password",
password));

// returna o objecto JSON
JSONObject json = jsonParser.getJSONFromUrl(loginURL,
params);
```

O método "*getJSONFromUrl"* da classe "*jsonParser"* recebe como parâmetros o endereço WEB servidor e uma lista de parâmetros, neste caso uma *tag*, *email* e *password*. Este método efetua o pedido HTTP ao servidor e retorna a resposta em formato JSON, conforme ilustra a Figura 29. O pedido HTTP é efetuado da seguinte forma:

```
// Pedido HTTP
DefaultHttpClient httpClient = new
DefaultHttpClient();

HttpPost httpPost = new HttpPost(url);
httpPost.setEntity(new UrlEncodedFormEntity(params));

// Envio dos dados
HttpResponse httpResponse =
httpClient.execute(httpPost);
```

A resposta do servidor WEB é obtida da seguinte forma:

```
// Obter resposta
HttpEntity httpEntity = httpResponse.getEntity();
is = httpEntity.getContent();
(…)
```

Assim o método "*doInBackgroud"* da *asynctask* é implementado da seguinte forma:

```
   @Override
   protected Integer doInBackground(Void... params)
   {
   String res = null;
   try {

   // método loginUser contido na classe userFunction

   UserFunctions userFunction = new UserFunctions();
   JSONObject json = userFunction.loginUser(email,
password);
```

O valor do objeto "*success"* da resposta do servidor, representados na Tabela 13**,** indica o sucesso ou insucesso do pedido:

```
   if (json.getString(KEY_SUCCESS) != null) {

   res = json.getString(KEY_SUCCESS);
```

Caso tenha o valor um, o que significa *login* efetuado com sucesso, os dados do utilizador são armazenados numa base de dados SQLite, interna ao sistema *Android*, com recurso ao método "*addUser"* da classe "*DatabaseHandler"*, que tem como parâmetros o nome, *email*, *uid* e data de criação.

```
   // Login com sucesso
   if (Integer.parseInt(res) == 1) {

   // Armazena dados do utilizador - SQLite Database
   // DatabaseHandler
   DatabaseHandler db = new
DatabaseHandler(getApplicationContext());

   JSONObject json_user =json.getJSONObject("user");

   // Limpar a base de dados
   userFunction.logoutUser(getApplicationContext());

   // Armazenar dados do utilizador
   db.addUser(json_user.getString(KEY_NAME),
              json_user.getString(KEY_EMAIL),
              json.getString(KEY_UID),
              json_user.getString(KEY_CREATED_AT));
   }
   (...)
   return Integer.parseInt(res);
   }
```

De seguida o método "*onPostExecute"* fecha a barra de progresso e é verificado o estado da variável "result" que indica o estado de *login*. Caso esta tenha o valor um, é iniciada a aplicação principal através de um *intent*. Se a variável "result" contiver o valor dois é apresentada uma mensagem de erro dado que o "*username"* ou "*password"* estão incorretos. No caso de a variável ter um outro valor, é apresentada a mensagem de erro "Erro de ligação", Figura 47.

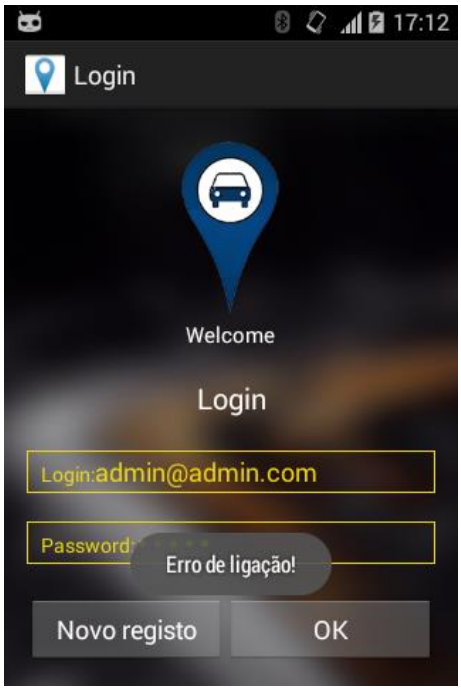


Figura 47 – Mensagem "Erro de ligação"

4.3.4. ATIVIDADE – REGISTO

A atividade de registo é composta por três caixas de texto editáveis, "nome", "email" e "password", (1, 2 e 3) e por dois botões, "Registar" e "Já registado. Entrar" (4 e 5), como visível na Figura 48. O funcionamento desta atividade é em tudo semelhante ao encontrado na atividade de *login*. A única diferença significativa é a chamada ao método "*registerUser"* em vez de ao método "*loginUser"* da classe "*userFunction"*, efetuado no método "*doInBackground"* da *asynctask "LoadRegisterTask*", no caso do botão de Registo seja premido.

O botão "Já registado. Entrar" direciona o utilizador novamente para a atividade de *login*. A Figura 49 ilustra o fluxo de dados da atividade.

O método "*registerUser"* tem como parâmetros os dados do utilizador (nome, *email* e *password)*. Este método cria uma lista de parâmetros que serão enviados ao servidor, e efetua o pedido HTTP com recurso ao método "*getJSONFromUrl"* da classe "*jsonParser"*, tal como realizado na atividade de *login*.

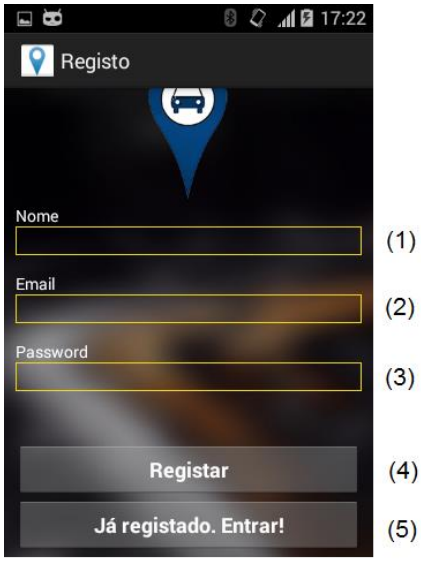


Figura 48 – Actividade de registo

A análise da resposta obtida do servidor fica a cargo do método "*onPostExecute"* da *asynctask*. No caso de a resposta ser de sucesso, a atividade principal "*Dashboard/OBDII"* é iniciada.

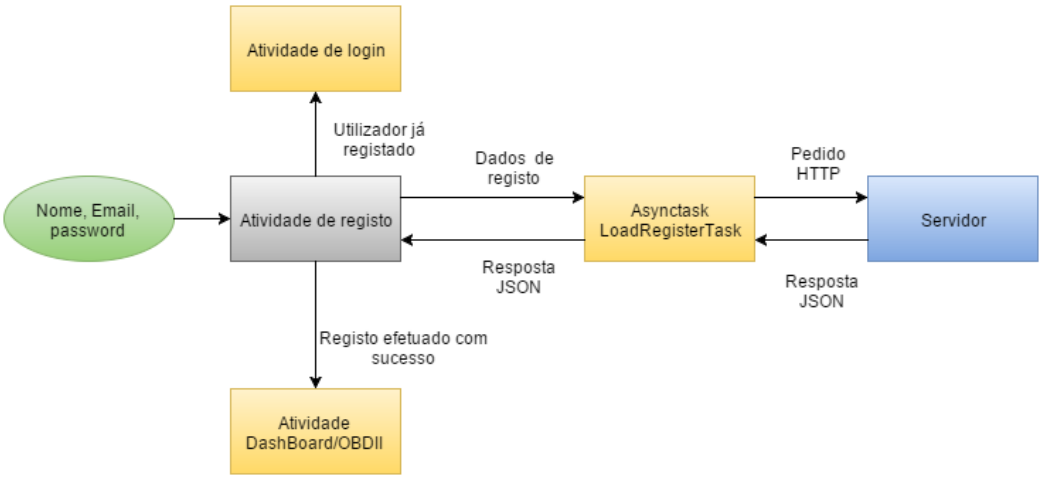


Figura 49 – Fluxo de dados da actividade de registo

Para outras respostas, e consultando a Tabela 12 que ilustra as possíveis respostas do servidor em modo de registo, são apresentadas ao utilizador mensagens de erro, correspondentes ao erro detetado.

4.4. DISPOSITIVO SENSOR BLUETOOTH

A aplicação de localização *Android* desenvolvida permite a comunicação com até sete sensores/dispositivos *Bluetooth* simultaneamente para obtenção de dados do veículo que não estejam disponíveis através do sistema OBDII.

Esta secção pretende demonstrar o desenvolvimento de um dispositivo que integra um sensor de temperatura e comunica com o *smartphone Android* através da tecnologia Bluetooth.

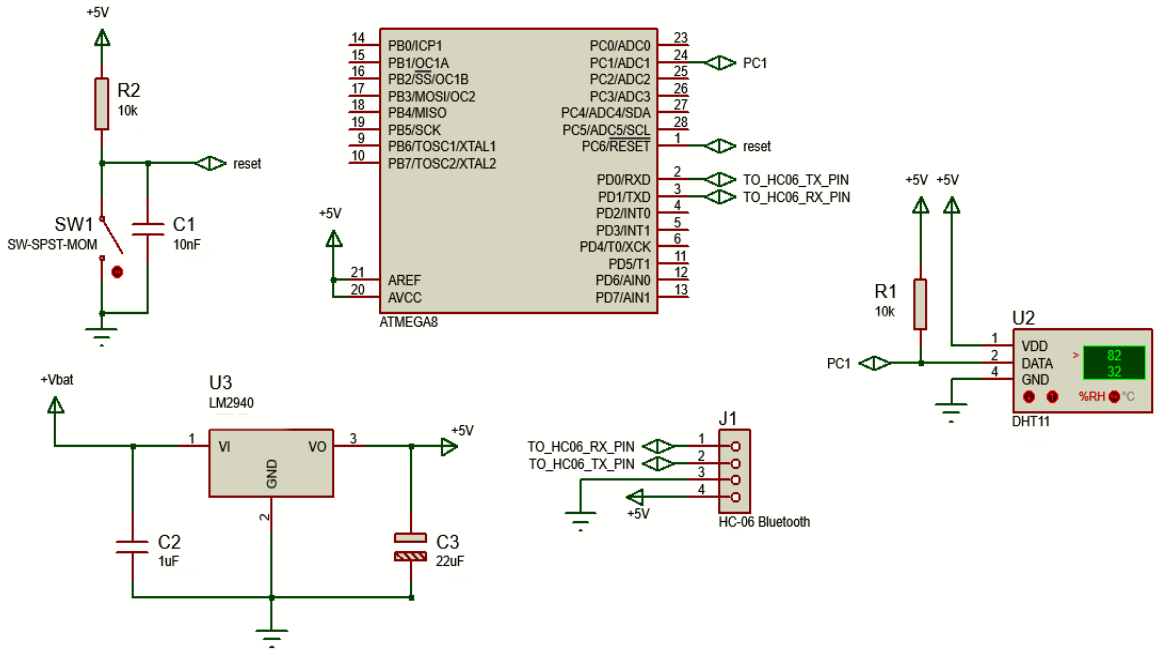


Figura 50 – Esquema elétrico do dispositivo desenvolvido

O dispositivo idealizado, Figura 50, é baseado num microcontrolador de 8 bits *Atmel AVR Atmega* 8. Este foi escolhido para o desenvolvimento do dispositivo dado as suas características, tais como:

- 8 Kb de memória Flash, 512 Bytes EEPROM e 1Kbyte de memória interna SRAM;
- 23 linhas de *input/output* (disponíveis em três portos: *Port B, Port C e Port D*);

- *Timer/Counter*: Três *timers* internos, dois de 8 bits e um de 16 bits;
- Comunicação SPI, USART e I2C;
- ADC com 10bits de resolução.

Apesar de este microcontrolador possuir algumas características não necessárias para o desenvolvimento deste dispositivo em particular, pode servir como base para outros dispositivos que possam no futuro ser desenvolvidos para integrar o sistema.

Assim, o dispositivo desenvolvido é composto por:

- Microcontrolador *Atmel AVR Atmega 8*;
- Circuito de alimentação;
- Circuito de *reset* para o microcontrolador;
- Sensor de temperatura/humidade DHT11;
- Módulo Bluetooth HC-06;

4.4.1. CIRCUITOS DE ALIMENTAÇÃO E *RESET*

O circuito de alimentação é baseado no integrado LM2940 fabricado pela *Texas Instruments.* Este integrado é um regulador linear de tensão destinado a alimentar o dispositivo e seus componentes (5 *volts*) a partir da linha de 12 *volts* existente no veículo. O LM2940 possui inúmeras vantagens face ao conhecido regulador de tensão 7805, entre as quais destacam-se:

- Desenhado para aplicações automóveis, com proteção de polaridade inversa e proteção contra picos de tensão.
- Regulador *Low Drop-Out* (LDO) – Permite que a tensão de entrada do integrado seja superior à de saída em apenas 0.5 volts (2 volts para o 7805). Um veículo ligeiro trabalha com uma tensão de alimentação entre 12 e 14 volts mas no momento de arranque do motor, a tensão da bateria do veículo pode descer a níveis muito baixos, por exemplo 6,5 volts. Nestes casos LM2940 consegue continuar a fornecer 5 volts na sua saída, dado que apenas necessita de 6 volts na sua entrada.

O circuito de *reset* permite reiniciar o microcontrolador, sem a necessidade de desligar o dispositivo da alimentação e voltar a ligar. O circuito é composto por uma resistência de *pull-up*, um interruptor *e* um condensador para filtrar possíveis ruídos de acionamento do interruptor.

4.4.2. SENSOR DHT-11

O dispositivo ilustrado na Figura 51, DHT-11, é um sensor de temperatura e humidade atmosférica de muito baixo custo, com uma saída digital de apenas um fio. Segundo a documentação do fabricante, o seu sensor de temperatura é composto por um termístor e o sensor de humidade é do tipo capacitivo [51]. A sua gama de medição é de 20 a 90% de humidade relativa e de 0-50ºC para a temperatura. A precisão de medição é de cerca de 5% para a humidade e 2% para a temperatura. Permite a alimentação numa gama de 3 a 5.5v.

Figura 51 – Sensor DHT-11 [51]

Este sensor possui três pinos de ligação: Alimentação (VCC), um pino de massa (GND) e o pino de saída de dados (DATA). A Tabela 23 ilustra as ligações efetuadas entre o microcontrolador e o sensor. O pino de saída de dados requer uma resistência de *pull-up* dado que o sensor possui uma saída do tipo coletor aberto.

Tabela 23 – Ligações entre o sensor DHT11 e o microcontrolador Atmega8

Atmega 8	Sensor DHT11
Pino 5v	Pino 5v
Pino Gnd	Pino Gnd
PC1 (*pull-up resistor*)	Pino Data

4.4.3. MÓDULO *BLUETOOTH* HC-06

O HC-06 é um módulo *Bluetooth* de classe 2 de baixo consumo e de baixo custo, Figura 52. É normalmente utilizado para adicionar conectividade *Bluetooth* a microcontroladores, dado que suporta o perfil SPP *Bluetooth*. O módulo HC-06 apenas pode funcionar como *slave*, ou seja, não pode iniciar uma ligação. Enquanto estiver ligado a outro dispositivo, o HC-06 funciona em modo transparente e não aceita comandos AT. As definições no módulo podem ser alteradas através de uma serie de comandos AT [52].

Figura 52 – Módulo Bluetooth HC-06

O módulo HC-06 vem pré configurado pelo fabricante da seguinte forma:

- *Bits per second: 9600bps*
- *Data bits: 8*
- *Parity: None*
- *Stop bits: 1*
- *Nome: linvor*
- *PIN: 1234*

No que diz respeito às ligações físicas, o módulo HC-06 utilizado possui quatro pinos, descritos na Tabela 24.

Tabela 24 – Pinos de ligação do módulo HC-06

Pino	Descrição
VCC	Alimentação (entre 3.3 volts e 5 volts)
GND	*Ground*
TXD	Saída de dados – (ligar ao pino RX do microcontrolador)
RXD	Entrada de dados (ligar ao pino TX do microcontrolador)

A Tabela 25 ilustra os comandos "AT" suportados pelo módulo HC-06, a resposta esperada e descrição de cada um dos comandos disponíveis.

Tabela 25 – Comandos AT do módulo HC-06 [52]

Comando AT	Resposta esperada	Descrição
AT	OK	Verificar comunicação
AT+VERSION	OKlinvorVX.X	Versão do *firmware*
AT+NAMExyz	OKsetname	Alteração do nome do módulo
AT+PIN1234	OKsetPIN	Alteração do PIN
AT+BAUD1	OK1200	*Baudrate*=1200bps
AT+BAUD2	OK2400	*Baudrate*=2400bps
AT+BAUD3	OK4800	*Baudrate*=4800bps
AT+BAUD4	OK9600	*Baudrate*=9600bps
AT+BAUD5	OK19200	*Baudrate*=19200bps
AT+BAUD6	OK38400	*Baudrate*=38400bps
AT+BAUD7	OK57600	*Baudrate*=57600bps
AT+BAUD8	OK115200	*Baudrate*=115200bps
AT+BAUD9	OK230400	*Baudrate*=230400bps
AT+BAUDA	OK460800	*Baudrate*=46800bps
AT+BAUDB	OK921600	*Baudrate*=921600bps
AT+BAUDC	OK1382400	*Baudrate*=1382400bps

4.4.4. PROGRAMAÇÃO DO MICROCONTROLADOR

O desenvolvimento do programa para o microcontrolador passa pela configuração da *Universal Synchronous Asynchronous Receiver Transmitter* (USART) para comunicação com o módulo *Bluetooth*/Aplicação *Android* e implementação do protocolo de comunicação do sensor DHT11, descrita na secção 4.4.2.

O algoritmo visível na Figura 53 ilustra o funcionamento geral do programa do microcontrolador. Apesar de o sensor utilizado conseguir dois parâmetros físicos (temperatura e humidade do ar), o programa irá utilizar apenas a leitura da temperatura do

ar, dado que a aplicação *Android* desenvolvida permite apenas receber um parâmetro por dispositivo sensor.

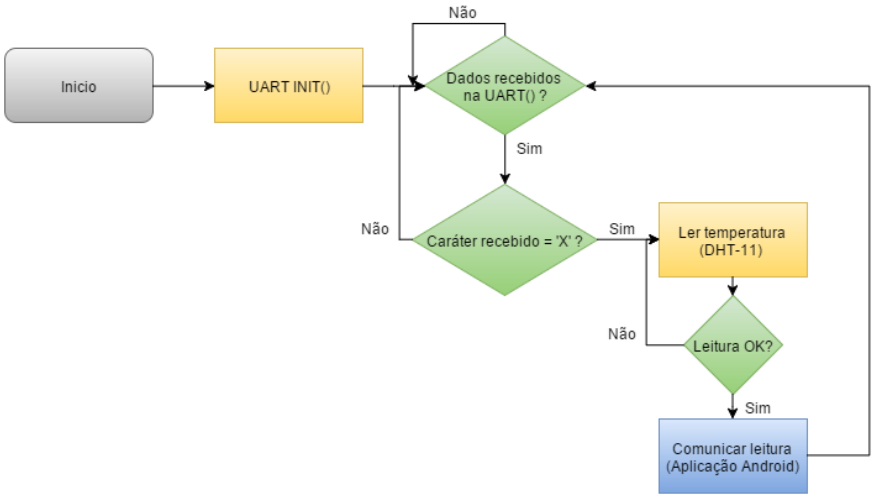


Figura 53 – Funcionamento geral do programa do microcontrolador

A. USART

A utilização da USART do microcontrolador AVR passa pela programação de cinco registros, de modo a coincidir com as configurações do módulo HC-06.

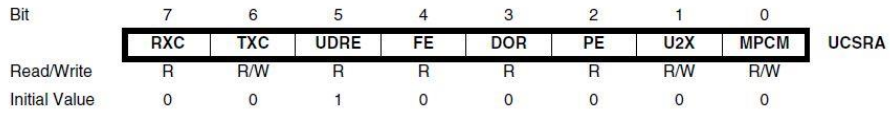
A descrição detalhada da funcionalidade de cada um dos *bits* dos registos pode ser consultada na *datasheet* do fabricante no microcontrolador [50] :

- *USART Data Register (*UDR) - Este registo contém dois *buffers*, Figura 54: O *Transmit Data Buffer Register* (TXB) é o destino dos dados escritos no registo UDR, enquanto a leitura do registo UDR retornará os dados contidos no *Receive Data Buffer Register* (RXB). Assim, a escrita e leitura deste registo permite enviar e/ou receber dados da aplicação *Android*, respetivamente;

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDR (Read)
	TXB[7:0]								UDR (Write)
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 54 – Registo UDR do microcontrolador AVR Atmega8

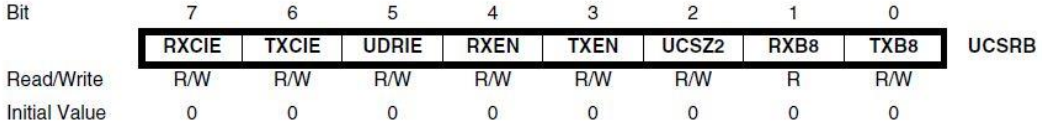
- *USART Control and Status Register A (*UCSRA) – Neste registo, Figura 55, será utilizado o *bit* RXC que indica que um *byte* está disponível (estado lógico "1") no registo UDR (recebido da aplicação *Android*) e o *bit* UDRE (estado lógico "1") que indica que um novo *byte* pode ser colocado no registo UDR para novo envio;



Bit	7	6	5	4	3	2	1	0	
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	UCSRA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

Figura 55 – Registo UCSRA do microcontrolador AVR Atmega8

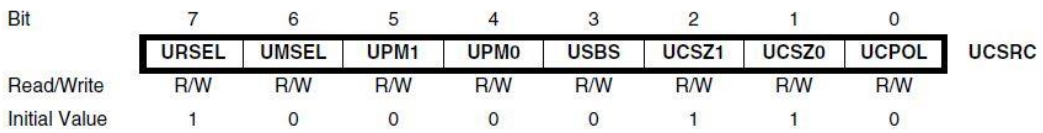
- *USART Control and Status Register B (UCSRB)* – A configuração deste registo, Figura 56, passa pela ativação dos elementos de receção e transmissão da USART, colocando num estado lógico "1" os *bits Receiver Enable* (RXEN) e *Transmitter Enable* (TXEN);



Bit	7	6	5	4	3	2	1	0	
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figura 56 – Registo UCSRB do microcontrolador AVR Atmega8

- *USART Control and Status Register C* (UCSRC) – Neste registo, Figura 57, é configurado o tamanho dos dados (8 *bits*) utilizado pelo transmissor/recetor definindo o *bit* UCSZ1 e UCSZ0 com o estado lógico "1". O *bit* UPM0 e UPM1 não são definidos, dado que os seus estados lógicos predefinidos ("0") desativam o modo de paridade. Os *bits* USBS e UMSEL cujas funções são definir o número de *stop bits* e modo de operação da USART, respetivamente, são deixados também com as configurações predefinidas (1 *stop bit* e modo assíncrono);



Bit	7	6	5	4	3	2	1	0	
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	UCSRC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	0	0	0	0	1	1	0	

Figura 57 – Registo UCSRC do microcontrolador AVR Atmega8

- ***USART Baud Rate Register (UBRR)*** – Este registo, permite definir a taxa de transmissão de dados. A taxa de transmissão por defeito do módulo HC-06 é de 9600bps. Assim e usando a fórmula disponível na *datasheet* do fabricante do microcontrolador é calculado o valor de UBRR.

$$UBRR = \frac{f_{osc}}{16 * Baud} - 1$$

Para $f_{osc} = 8\,Mhz$ e $Baud = 9600bps$ o valor obtido para UBRR é 51.

A Figura 58 ilustra o registo UBRR do microcontrolador ATMEL AVR Atmega8.

Bit	15	14	13	12	11	10	9	8	
	URSEL	–	–	–	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Figura 58 – Registo UBRR do microcontrolador AVR Atmega8

B. Comunicação com o sensor DHT-11

O sensor DHT-11 numa transmissão completa envia 40 *bits* de dados com o seguinte formato [51]:

"*8bit integral RH data + 8bit decimal RH data + 8bit integral T data + 8bit decimal T data + 8bit check sum*".

Se a transmissão for completada com sucesso, o *check sum, que* é um código usado para verificar a integridade de dados transmitidos, terá de ser igual à seguinte soma:

"8bit integral RH data + 8bit decimal RH data + 8bit integral T data + 8bit decimal T data".

A comunicação com este sensor é baseado num diagrama temporal conforme ilustra a Figura 59. Assim a comunicação inicia-se com o microcontrolador a definir o pino de ligação ao

DHT11 como *output* e a colocar a linha de dados do DHT11 num nível lógico "0", pelo menos durante 18 milissegundos, para o sensor ter tempo de detetar esta alteração lógica.

De seguida coloca esta linha de novo num nível lógico "1" e espera entre 20 a 40 microssegundos por uma resposta do sensor. Para receber a resposta do sensor o microcontrolador redefine o pino de ligação ao DHT11 como *input.*

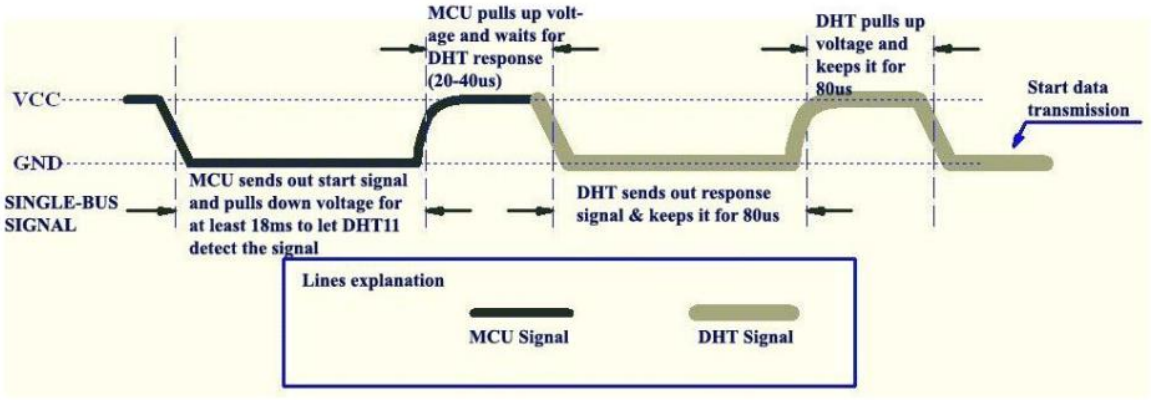


Figura 59 – Diagrama temporal - DHT11 [51]

O sensor deverá responder, colocando a linha de dados num nível lógico "0" durante 80 microssegundos e posteriormente de novo num nível lógico "1" durante 80 microssegundos. O próximo estado logico "0" enviado pelo sensor, é o início da transmissão dos dados a receber.

Após este estado lógico "0", o sensor deve colocar a linha de dados num estado lógico "1" durante um determinado tempo. Se a linha de dados estiver durante 26 a 28us num estado lógico "1", isto representa que o *bit* transmitido é um "0", como ilustra a Figura 60.

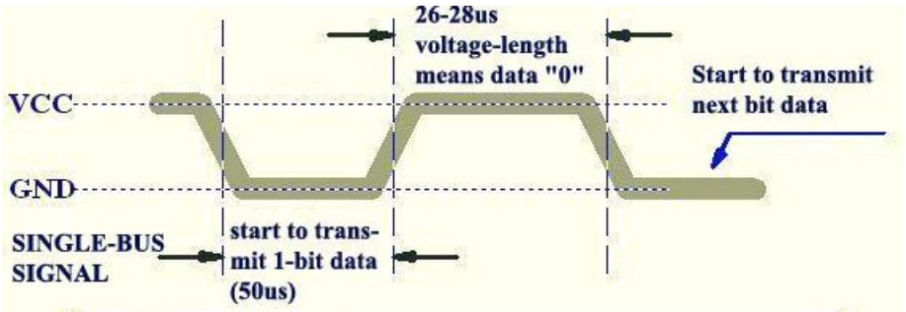


Figura 60 – DHT11 – Representação do nível lógico "0" [51]

No caso da linha de dados do DHT11 estiver até 70 microssegundos num estado lógico "1", isto representa que o *bit* transmitido é um "1", como ilustra a Figura 61.

Deste modo é possível obter as leituras de humidade e temperatura enviadas pelo sensor.

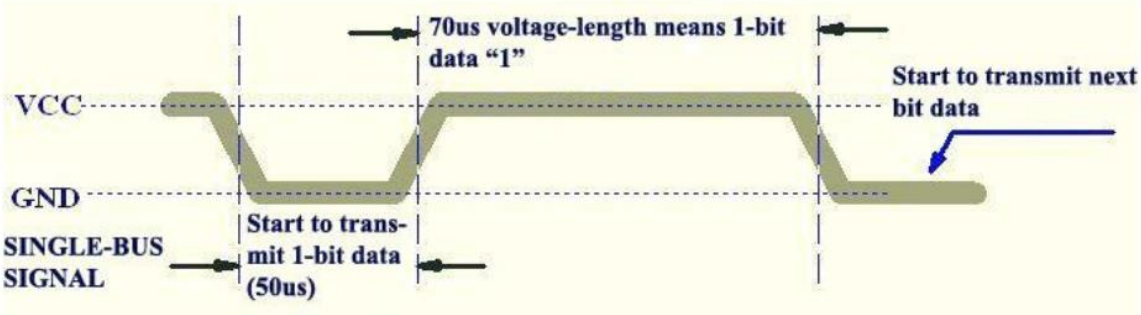


Figura 61 – DHT11 - Representação do nível lógico 1 [51]

4.5. APLICAÇÃO WEB

Esta secção pretende demonstrar o desenvolvimento da aplicação WEB.

4.5.1. COMUNICAÇÃO COM APLICAÇÃO *ANDROID*

Para a troca de dados entre a aplicação *Android* e uma base de dados Mysql foi utilizada uma API, descrita na secção 3.4.3.

Esta API é composta por quatro ficheiros PHP: "Config", "DB_connect", "DB_functions" e "Index", implementando apenas os métodos correspondentes ao sistema de *login* e registo, sendo que os todos os restantes métodos necessitaram de ser implementados.

A. Ficheiro Index.php

O ficheiro "index.php*"* aceita os pedidos remotos enviados pela aplicação *Android* e enviará uma resposta como ilustra a Figura 29. Este ficheiro começará por analisar a "tag" de identificação dos dados enviada pela aplicação *Android*.

Dependendo da "tag" recebida, um dos métodos da classe "DB_Functions" será chamado, que fará uma operação na base de dados e retornará uma resposta, Figura 62.

A Tabela 26 descrimina quais os métodos utilizados.

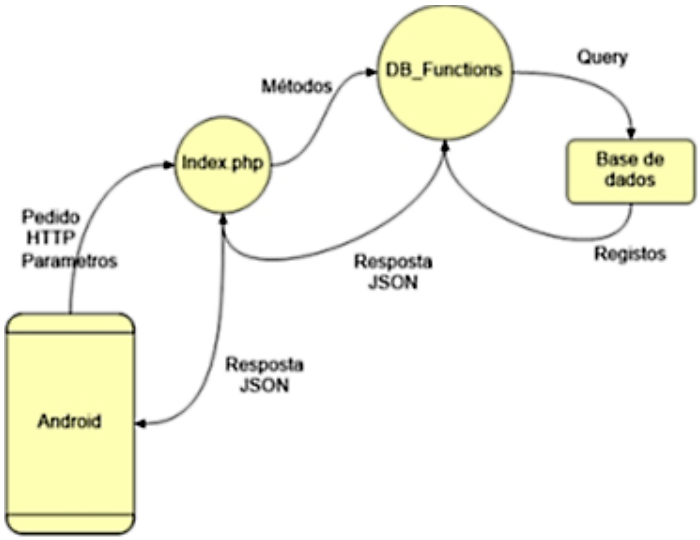


Figura 62 – Fluxo de dados entre a aplicação *Android* e o servidor

Tabela 26 – Métodos da classe BD_Functions utilizados

TAG	**Parâmetros enviados pela aplicação Android**	**Métodos da classe DB_Functions**	**Descrição**
Login	Tag, Email,Password	getUserByEmailAndPassword	Verifica utilizador na BD
register	Tag, Nome, Email, Password	storeUser	Armazena utilizador na BD
addlocationFULL	Tag, JsonArray	- storeLocationFull - storeLocationOBD - storeLocationSensores - storeLocationOnly	Armazena localização na BD

A aplicação "index.php" começa por verificar o conteúdo da variável "tag". Caso esta exista e não seja "null" a variável é lida.

Posteriormente é verificada qual a" tag" enviada pela aplicação *Android*. Dada a extensão do código utilizado, será analisado apenas a principal funcionalidade do sistema desenvolvido (um exemplo do procedimento realizado como resposta a um envio de localização pela aplicação *Android*).

De seguida para cada elemento do *array* (de localização) são obtidos os valores enviados pela aplicação de localização. São definidas também duas *flags*, para a sinalização da presença de valores do adaptador OBDII/Bluetooth e sensores Deste modo é possível posteriormente, chamar o método correto da classe "DB_Functions" para guardar os dados correspondentes.

A presença do elemento "faultcode" indica a presença de valores OBDII no elemento do *array*, sendo colocada a *flag* "flag_OBD" a "true" e lidos todos os valores:

Da mesma forma, a presença do elemento "sensor_name", indica a presença de sensores nos dados de localização.

Posteriormente e consoante o estado das duas *flags* de sinalização é chamado o método correto da classe "DB_Functions". Dependendo do retorno do método utilizado, é enviada a resposta para a aplicação *Android*. A resposta recebida (JSON) pela aplicação *Android* será em caso de sucesso:

```
{
"tag":"addlocationFULL",
"success":1,
"error":0
}
```

B. Ficheiro de interação com a base de dados - DB_Functions

A classe "DB_Functions" implementa os métodos utilizados pelo ficheiro "index.php" para comunicação com a base de dados Mysql. A esta classe da API utilizada, foram adicionados os seguintes métodos:

- "storeLocationFull" – Armazena uma localização (dados GPS, sensores e OBDII);
- "storeLocationOBD" - Armazena uma localização (dados de GPS e OBDII);

- “storeLocationSensores” - Armazena uma localização (dados de GPS e sensores);
- “storeLocationOnly” - Armazena uma localização (dados de GPS).

Dada a semelhança do código produzido para os quatro métodos, será descrita a implementação apenas do método “storeLocationFull” que armazena na base de dados uma localização que contem dados de GPS, sensores e OBDII.

A implementação do método “storeLocationFull” da classe “DB_Functions” inicia-se pela tentativa de inserção dos dados provenientes do recetor GPS na tabela “locations”. Se o registo for inserido com sucesso é obtido o seu “id” de registo.

Este “id” atribuído ao campo “location_ID” das tabelas “sensordata”, “obddata” e “faultcodes”, permite associar os registos destas tabelas a um registo da tabela “locations”.

No que diz respeito aos DTC’s do veiculo, estes são enviados pela aplicação *Android* no formato “[Xxxxx, Xxxxx, Xxxxx]”. Assim torna-se necessário separar cada um dos DTC, de modo a que cada DTC seja inserido como um único registo na tabela “faultcodes”.

De modo a eliminar os caracteres delimitadores é utilizada a função “trim” e posteriormente cada um dos DTC é colocado numa posição de um *array (*função *explode)*, de modo a inserir cada DTC como um registo na tabela “faultcodes”.

Posteriormente são inseridos os dados provenientes dos sensores *Bluetooth*. Neste caso a aplicação *Android* envia os dados dos sensores no seguinte formato:

- Nome dos sensores:

```
"[nomesensor1,nomesensor2,…] "
```

- Valor dos sensors:

```
  "[sensor1type;sensor1units;sensor1value,
sensor2type;sensor2units;sensor2value,...] "
```

Deste modo são necessárias algumas operações de formatação de forma a separar os dados provenientes de cada um dos sensores, utilizando as funções PHP “trim” e “explode”.

C. Ficheiro de ligação “DB_Connect.php” e ficheiro de configuração “config.php”

O ficheiro “DB_Connect.php” possui os métodos de ligação à base de dados “mysql_connect” e “mysql_close”. Faz também a seleção da base de dados através do método “mysql_select_db”.

O ficheiro “config.php” define variáveis contendo o endereço do servidor SQL, dados de autenticação e o nome da base de dados.

D. Base de dados MYSQL

O modelo da base de dados MYSQL implementado no servidor é semelhante ao criado para a base de dados da aplicação *Android*, Figura 63. A única diferença entre as duas é a tabela “users”.

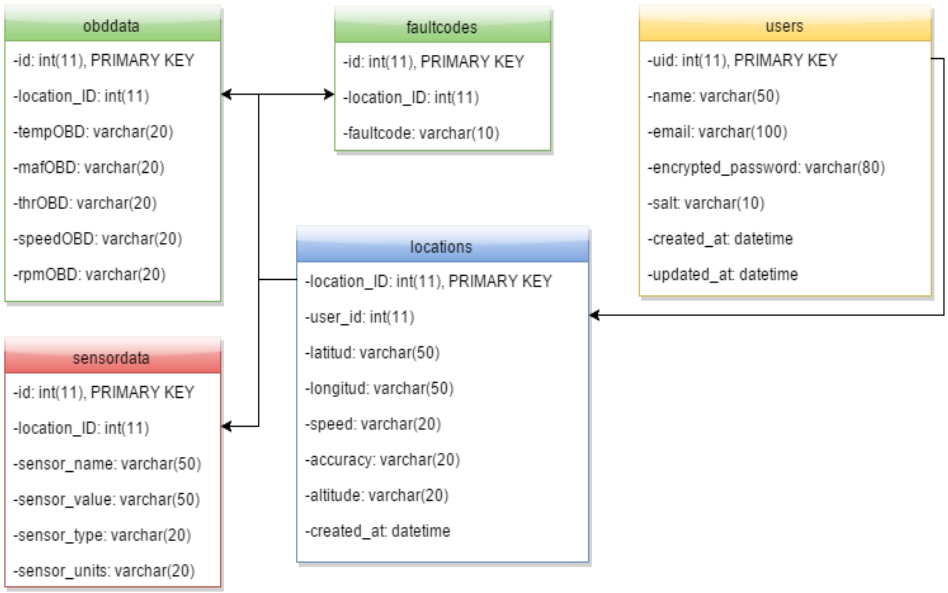


Figura 63 – Modelo físico da base de dados MYSQL

4.5.2. APLICAÇÃO WEB PARA VISUALIZAÇÃO DE DADOS

Para um utilizador/gestor verificar onde está cada veículo geograficamente, é necessário uma aplicação que permita mostrar estas informações num mapa. Para além disso, e dado que o

sistema permite que cada veículo possa ser equipado com um adaptador OBDII/Bluetooth e sensores, torna-se indispensável fornecer estes dados adquiridos ao utilizador.

A solução encontrada passa por implementar um sistema baseado numa aplicação/página WEB. A aplicação é composta por:

- Página "login" – Autenticação no sistema;
- Página "Latest Vehicles Location*"* – Mostra ao utilizador um mapa (*Google Maps*) com a última localização de cada veículo registado no sistema;
- Página "Vehicle Location history" – Mostra ao utilizador um mapa com as localizações de um determinado veículo;
- Página "OBDII Data" – Gráfico com informação as OBDII recolhidas de um determinado veículo;
- Página "Sensor Data*"* – Tabela com os dados recolhidos de cada sensor de um determinado veiculo;
- Página "Logout*"* – Encerra sessão do utilizador no sistema.

A Figura 64 ilustra o funcionamento geral e componentes da página WEB. O utilizador inicia a interação com o sistema através da página "login", onde fará a autenticação no sistema.

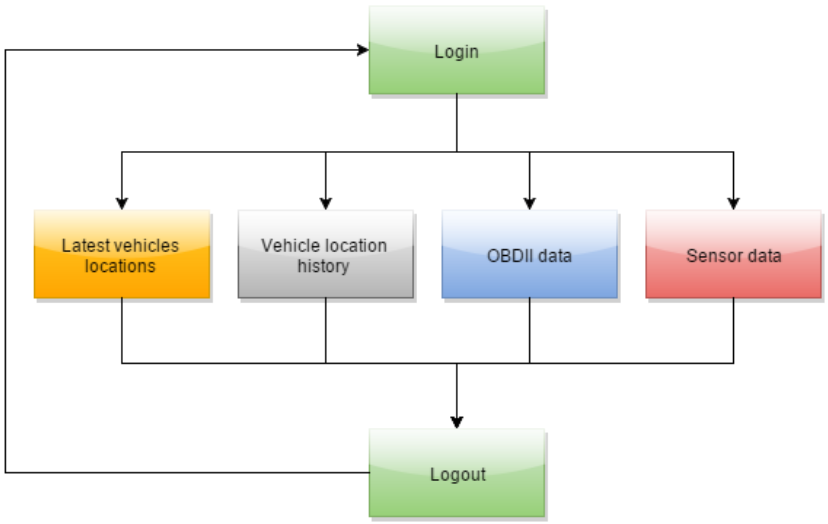


Figura 64 – Componentes da página WEB

Após a autenticação o utilizador terá acesso a quatro diferentes páginas, onde poderá consultar os dados pretendidos. Após esta consulta, se desejar poderá sair do sistema através da página "logout".

A. Página de *Login*

A página *login* dispõem de um formulário HTML para o utilizador introduzir os seus dados de autenticação (*email* e *password*), Figura 65.

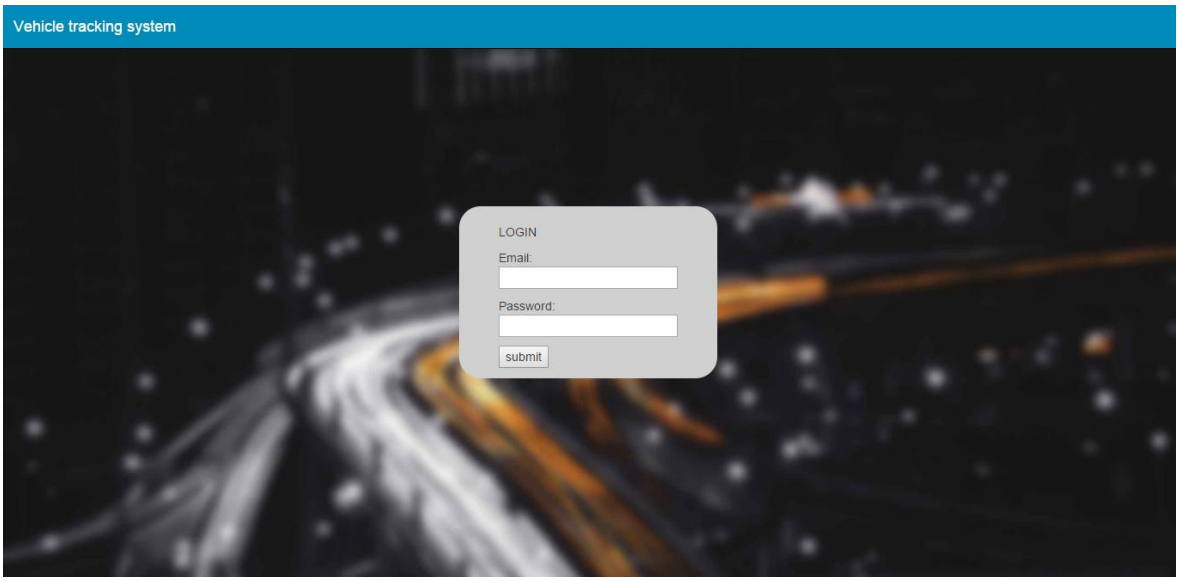


Figura 65 – Página de *login*

De forma a permitir também aos utilizadores da aplicação *Android* acesso ao sistema WEB, os dados de autenticação são os mesmos para os dois sistemas.

O sistema de *login* passa pela utilização de variáveis de sessão e pela comparação entre os dos dados de autenticação introduzidos pelo utilizador no formulário HTML e os dados contidos na base de dados. Uma sessão é uma forma de armazenar informações em variáveis, para não ser necessário a autenticação do utilizador em múltiplas páginas. As variáveis de sessão armazenam assim informações do utilizador, como o nome de utilizador ou *password*, entre outras. Por defeito, as variáveis de sessão duraram até ao fecho do navegador WEB (*browser*).

O primeiro passo para a implementação do sistema de *login* é verificar se na abertura da página existe uma sessão já iniciada.

Após recorrer à função "session_start()", posteriormente é verificado se as variáveis de sessão *email* e *password* estão definidas. Caso não estejam definidas a sessão é destruída (função "session_destroy()") . No caso de as variáveis estarem definidas, o utilizador é redirecionado para a página "Latest Vehicles Location " (maps_latest.php). No caso de o utilizador não estar ainda autenticado, será necessário o preenchimento do formulário com os respetivos dados de autenticação.

Após o envio do formulário de autenticação (botão *submit*) os dados introduzidos são comparados com os dados disponíveis na tabela "users" da base de dados MYSQL, através do método "getUserByEmailAndPassword()" da classe "DB_Functions". Este método retorna "true*"* ou "false*"* consoante a correspondência dos dados. Caso exista correspondência, os dados introduzidos pelo utilizador são colocados em variáveis de sessão e este é reencaminhado para a página "Latest Vehicles Location".

B. Página "Latest vehicles locations"

A página "Latest Vehicles Location*"* é responsável por mostrar ao utilizador um mapa (*Google Maps*) com a última localização de cada veículo registado no sistema, Figura 66.

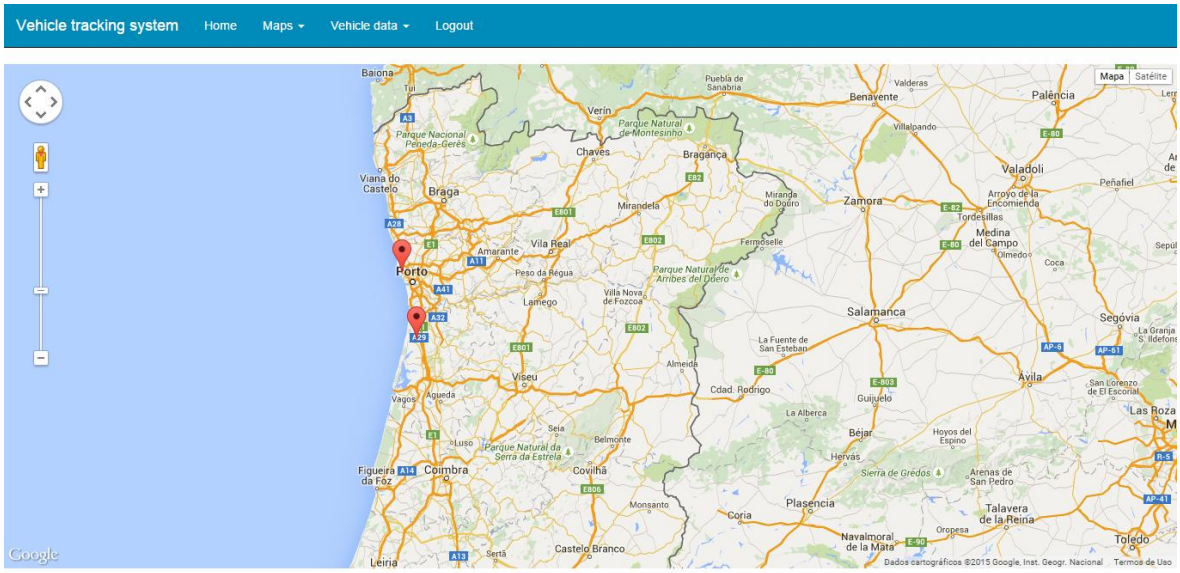


Figura 66 – Página "Latest Vehicles Location*"*

Um marcador vermelho é introduzido na posição geográfica de cada um dos veículos.

Se o utilizador carregar em qualquer um dos marcadores surgirá uma pequena janela (*infoWindow*) com as informações atuais do veículo (utilizador, dados OBDII e sensores) como ilustra a Figura 67.

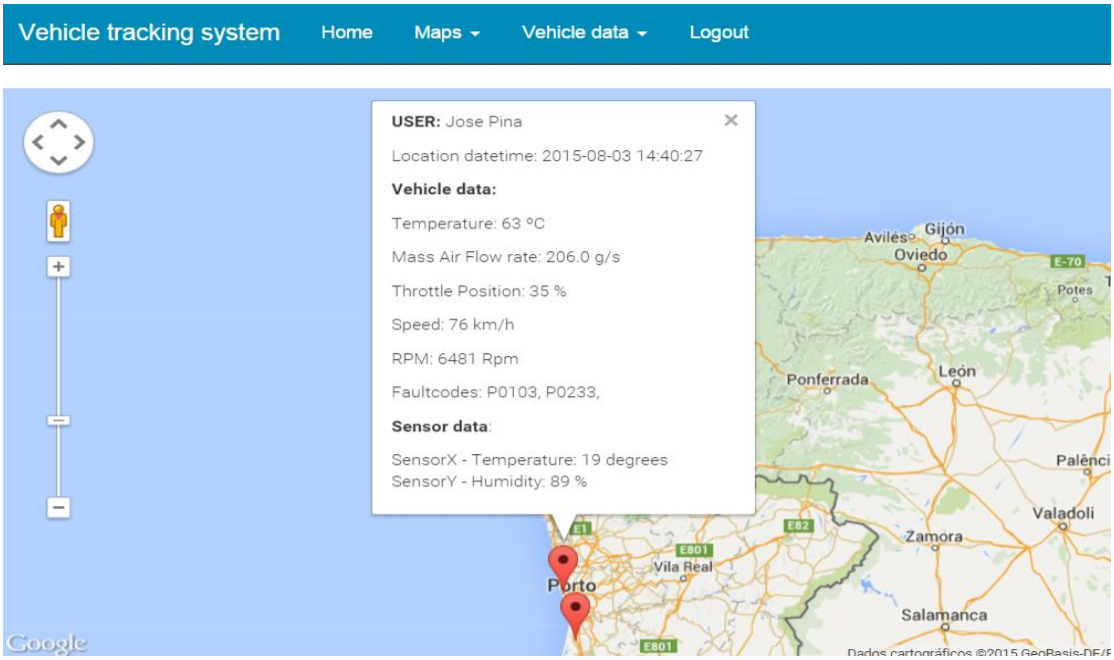


Figura 67 – Página "Latest Vehicles Location*"* - *InfoWindow*

O Google fornece no seu guia de desenvolvimento do *GoogleMaps*, um exemplo de como pode ser feita a aquisição de dados de uma base de dados MYSQL, utilizando PHP, para posterior uso dos dados com o *GoogleMaps*. Este exemplo foi seguido e implementado com modificações, de modo a poder corresponder ao tipo de dados existente neste trabalho e funcionalidades esperadas [44].

O primeiro passo na implementação passa por definir um evento *onload* no elemento "body" da página WEB, de modo a ser executado um script (Javascript) após o carregamento da página. Este script "load()" é responsável por inicializar o mapa (definindo opções como o zoom inicial e tipo de mapa) e pedir os dados dos veículos à base de dados.

A função "downloadUrl" faz o *download* dos dados dos veículos da base de dados MYSQL através de uma chamada ao script PHP "phpsqlajax_genxml.php". Os dados obtidos por este serão passados à função "xmlprocessing", como ilustra a Figura 68.

Esta função “downloadUrl” faz uso do objecto “XMLHttpRequest”, que é utilizado para a troca de dados com um servidor em *background*, permitindo assim por exemplo:

- Atualizar a página web sem ser necessário recarregar a página;
- Solicitar dados de um servidor após a página ser carregada;
- Receber dados de um servidor após a página ser carregada;
- Enviar dados para um servidor em *background*.

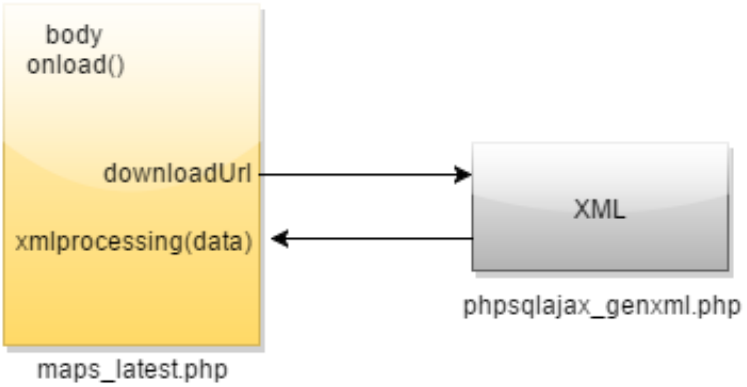


Figura 68 – *Download* dos dados dos veiculos da base de dados MYSQL

Assim a implementação da função “downloadUrl” inicia-se pela criação um novo objeto “XMLHttpRequest” e de um *event listener* “onreadystatechange”, que reporta alterações no estado do pedido através da propriedade “readyState”.

Os possíveis estados do “XMLHttpRequest” lidos através da propriedade “readyState” são ilustrados na Tabela 27.

Tabela 27 – Estados do XMLHttpRequest

Propriedade	**Estados do XMLHttpRequest (alterado automaticamente)**
readyState	0: pedido não inicializado 1: Ligação ao servidor estabelecida 2: Pedido recebido 3: Processando pedido 4: Pedido concluído e a resposta está disponível

Para enviar o pedido para o servidor são utilizados os métodos "open()" e "send()" do objeto "XMLHttpRequest". A Tabela 28 descreve os dois métodos.

Tabela 28 – Métodos do objecto XMLHttpRequest

Método	Descrição
open(método,url,*async*)	Especifica o tipo de pedido, URL e se o pedido deve ser tratado de forma assíncrona ou não: Método: Tipo de pedido: GET ou POST Url: Localização do ficheiro no servidor *async*: true (assíncrono) ou false (síncrono)
send(*string*)	Envia pedido para o servidor: *string*: Apenas para pedidos POST

Como descrito anteriormente, o *script* "phpsqlajax_genxml.php" é o destinatário do pedido. A função deste *script* PHP é recolher a última localização de cada veículo registado no sistema, e gerar um ficheiro XML com essas informações.

Para gerar um ficheiro XML é utilizada a biblioteca *Document Object Model* (DOM) fornecida na instalação do PHP. A biblioteca DOM permite ler e escrever um documento XML ou HTML e representa-lo como uma árvore de nós.

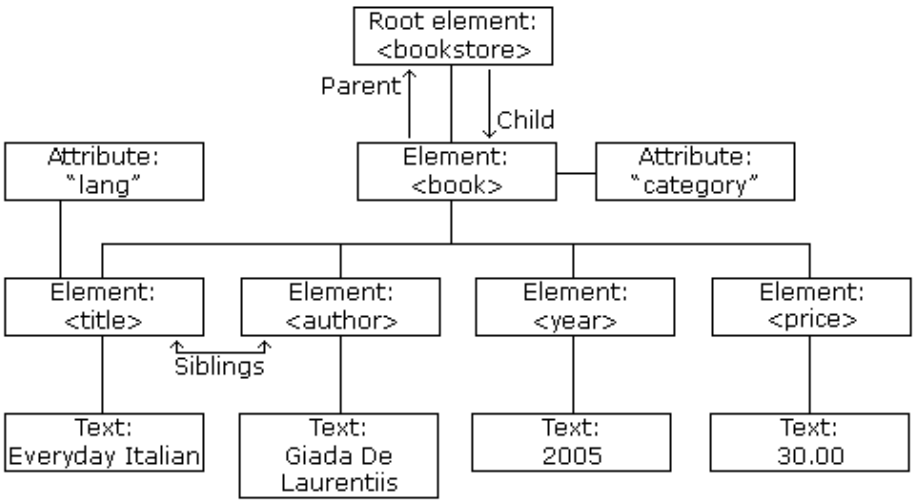


Figura 69 – Arvore DOM XML

A Figura 69 apresenta um exemplo de uma árvore DOM (XML). O nó raiz é neste caso "bookstore", que tem como nó filho o elemento "book". O nó "book" tem quatro nós filhos, "title", "author","year" e "price" com um nó "text" cada um. Cada nó poderá depois possuir atributos conforme necessário [45].

O DOM XML, utilizado neste caso, define objetos e propriedades para todos os elementos XML e os métodos de acesso. Assim o DOM XML interpreta um ficheiro XML como uma estrutura em árvore.

Alguns métodos utilizados:

- "x.getElementsByTagName(name)" – obter todos os elementos com a tag "name".
- "x.appendChild(node)" – associar um nó filho ao nó x
- "x.removeChild(node)" - remover um nó filho do nó x

A implementação deste *script* começa pela criação de um objecto "DOMDocument". De seguida é feita a ligação à base de dados MYSQL:

```
   <?php
   // Dados de acesso à base de dados
   require("phpsqlajax_dbinfo.php");

   // Criar ficheiro XML, Nó raiz

   $dom = new DOMDocument("1.0");

   // Abrir ligação com a base de dados MYSQL
   $connection=mysql_connect ('localhost', $username,
$password);
```

Posteriormente é selecionada a base de dados e feito o pedido SQL.

```
   (...)

   // Selecionar a base de dados
   $db_selected = mysql_select_db($database,
$connection);
   (...)
   // Pedido SQL (última localização de cada utilizador)

   $query = "SELECT t1.* FROM locations t1 WHERE
t1.created_at = (SELECT MAX(t2.created_at) FROM
locations t2 WHERE t2.user_id = t1.user_id )";

   $result = mysql_query($query);
   if (!$result) {die('Invalid query: ' .
mysql_error());}
```

De seguida é construída a árvore de nós representativa dos dados obtidos do servidor, como ilustra a Figura 70.

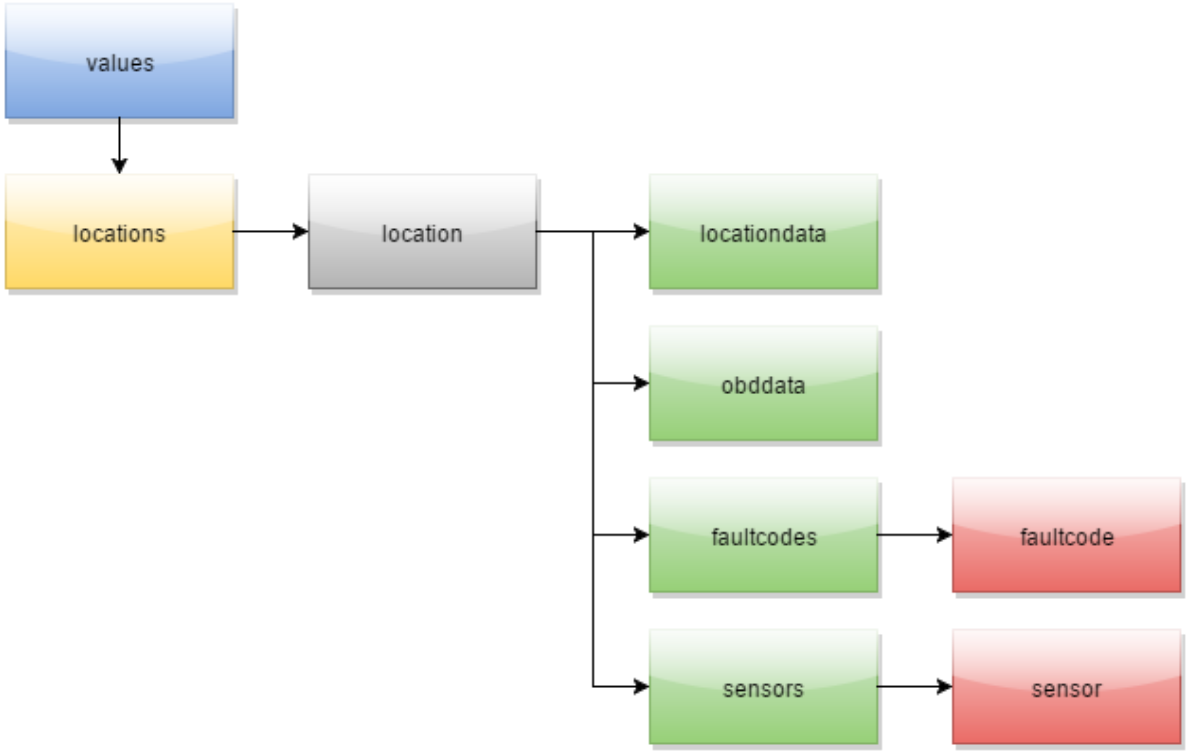


Figura 70 – Árvore de nós gerada pelo *script* PHP (*phpsqlajax_genxml.php*)

Esta árvore de nós é criada com recurso aos métodos "createElement" e "appendChild" do objeto "DOMDocument", que permitem criar os diferentes elementos que compõem a árvore e respetiva hierarquia, como ilustra a Figura 70 [44][45][46] .

De forma a replicar essa estrutura, o primeiro passo é criar o nó raiz "values":

```
header("Content-type: text/xml");

  // Nó raiz "values"
  $root = $dom->createElement("values");
  $dom->appendChild($root);
```

De seguida é criado o nó "locations", nó "location" e o nó "locationdata", sendo preenchidos com os dados obtidos da base de dados MYSQL. Após a obtenção dos dados de localização (GPS), é consultada a tabela "faultcodes" da base de dados e é criado o nó "faultcodes" que armazenará os DTC's do veículo nessa localização (se possuir).

A próxima etapa passa pela consulta da tabela "obddata", de modo a obter os dados OBDII do veículo nessa localização. Por fim a tabela "sensordata" é consultada e após o preenchimento do nó "sensors" e "sensor" com os respetivos atributos é gerado o ficheiro XML através do método *saveXML*. Um exemplo de um possível resultado obtido é ilustrado na Figura 71.

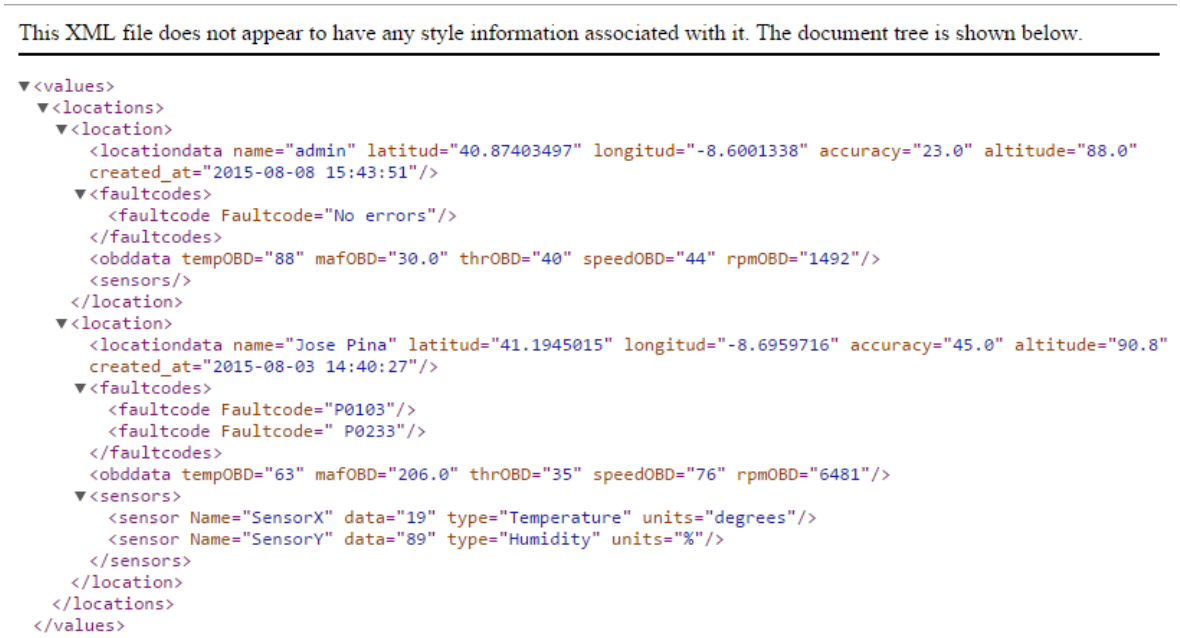


Figura 71 – Ficheiro XML gerado pelo script PHP (*phpsqlajax_genxml.php*)

Após este procedimento, a função "xmlprocessing" do *script* recebe estes dados formatados em XML.

Para apresentação destes dados no mapa é necessário:

- Separar dados por utilizador;
- Adicionar um marcador na localização de cada utilizador;
- Associar uma janela (*InfoWindow*) a cada marcador, de modo a permitir a visualização dos dados do veículo nessa localização.

Para ler cada nó do ficheiro XML utiliza-se o método "documentElement.getElementsByTagName" e para ler os atributos o método "getAttribute". Assim a função começa por ler a resposta do servidor e irá ler o conteúdo de cada nó "location" que corresponde à localização de dado um utilizador. Para cada nó

"location" a função necessita de ler o conteúdo de todos os nós filhos, "locationdata", "obddata", "faultcodes", "sensors". A leitura do nó "locationdata" permite obter o nome do utilizador, coordenadas de localização GPS, acurácia e data e hora em que o veiculo se encontrava nessa localização.

A leitura do nó "obddata" permite obter os dados OBDII do veículo e DTC's. De forma a verificar se o nó "obddata" contem dados é utilizado o método "hasAttributes()", que retorna "*true"* ou "*false"* caso este possua ou não atributos. Isto permite também definir a *flag* de sinalização "OBD=true", para posterior verificação na definição da janela associada ao marcador. Por fim é apenas necessário ler o nó relativo aos DTC's e sensores. Na leitura do nó "sensors" é também utilizada uma *flag* de sinalização (SENS).

Tendo obtido os dados que compõem cada marcador e de forma a construir a janela associada a cada um dos marcadores é necessário antes de mais definir o seu conteúdo. Este conteúdo é composto por código HTML. Fazendo uso das duas *flags* de sinalização de presença de dados OBDII e sensores, é possível apresentar diferentes mensagens ao utilizador dependendo do tipo de dados presentes. No caso de por exemplo um dos elementos não estar presente é mostrada a mensagem "No available data!" no respetivo campo. O mesmo se passa para os restantes casos.

Posteriormente é criado o marcador e é definida a janela (*InfoWindow*) que está associada ao marcador, através da função "bindInfoWindow".

A função "bindInfoWindow" cria um *click event listener* para o marcador, cuja função é apresentar a janela ao utilizador. Através dos métodos "setContent" e "open" da "InfoWindow", o conteúdo da janela é definido com o código HTML anteriormente especificado e esta é apresentada ao utilizador.

Uma função "setTimeout" é utilizada para agendar a execução da próxima tarefa de aquisição de dados, de modo a atualizar os marcadores existentes do mapa.

C. Página *Vehicle location history*

A página *Vehicle location history* apresenta ao utilizador/gestor todas as localizações de um dado utilizador da aplicação *Android* de localização, num intervalo de tempo à escolha

(formulário HTML). Para além disso, exibe os valores médios e máximos dos parâmetros OBDII obtidos, bem como os DTC's existentes, Figura 72. A sua implementação é semelhante à da página *Latest vehicles locations*.

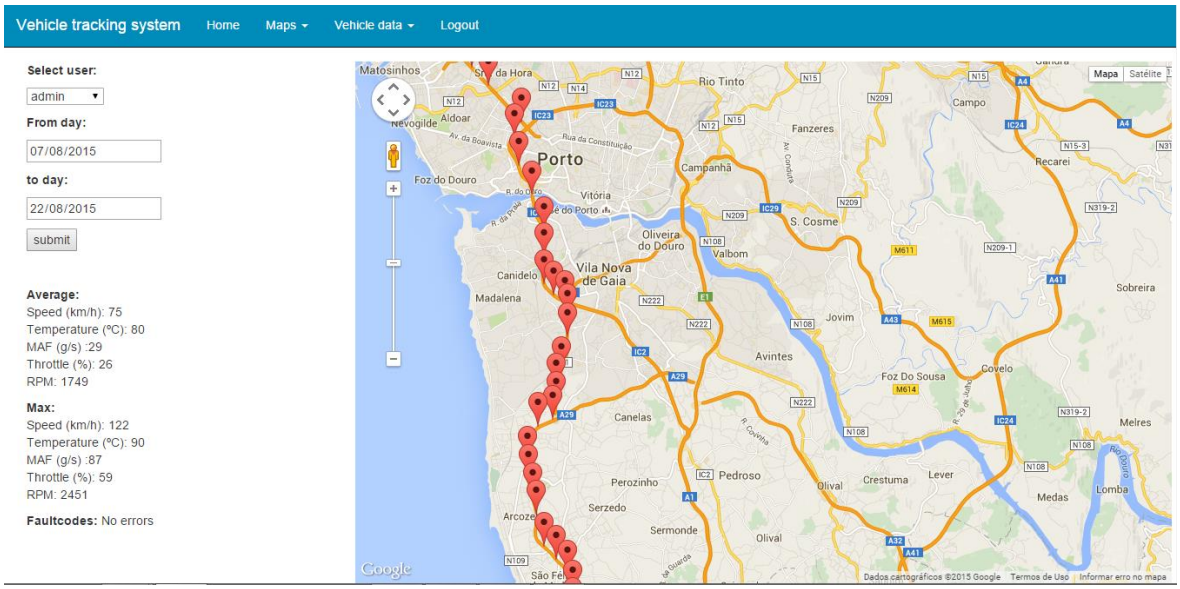


Figura 72 – Página *Vehicle location history*

A principal diferença na implementação, é a chamada à função "downloadUrl", responsável por executar o script PHP "phpsqlajax_genxml_unique.php" com os dados do formulário como parâmetro. Assim, o script PHP irá construir a sua *query* SQL com estes parâmetros, de forma a obter os resultados pretendidos para a operação.

No *script* PHP e de forma a calcular, por exemplo, o valor máximo de um dos parâmetros OBDII, cada valor obtido é colocado num *array*. Posteriormente é verificado qual o maior valor presente nesse *array* utilizando a função "Math.max.apply()". Dado que cada campo onde os resultados serão apresentados ao utilizador, são identificáveis, o valor correspondente a esse campo pode ser modificado utilizando *jquery* com recurso ao método ".html()".

D. Página OBDII data

A página "OBDII data", ilustrada na Figura 73 apresenta os valores OBDII de um veículo num gráfico. Esta página fornece um formulário, que permite ao utilizador/gestor escolher

de qual utilizador/veículo pretende visualizar dados OBDII e qual o intervalo de tempo pretendido. Dado que dependentemente do intervalo de tempo pretendido, o gráfico terá um tamanho variável, é possível que o gráfico se torne praticamente ilegível.

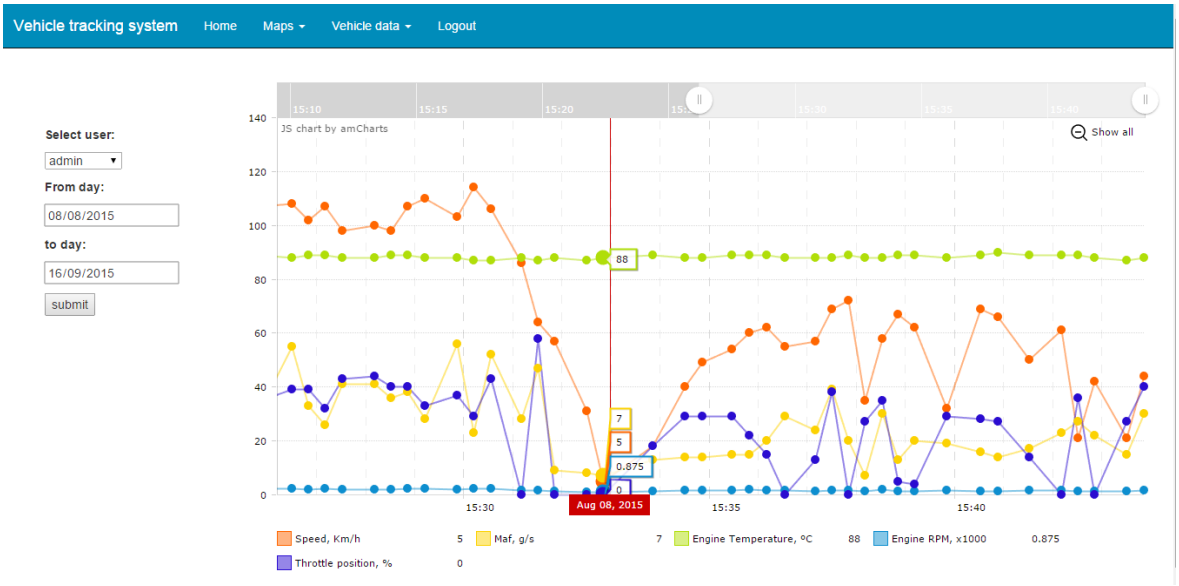


Figura 73 – Página "OBDII data"

Desta forma, foi utilizada a biblioteca "amCharts" que torna possível a implementação de gráficos dinâmicos [47]. A biblioteca utiliza as tecnologias JavaScript/HTML5. Algumas das características decisivas na escolha desta biblioteca:

- *Zoom/Pan* no gráfico
- Desativar/ativar gráficos
- Possibilidade de apresentar grandes quantidades de dados
- Formatação de gráficos com base em data/hora

No que diz respeito à implementação e após a tradicional definição do formulário HTML e *layout* da página, é definido o *script* responsável pela inicialização do gráfico (JavaScript). Este *script* começa por definir o evento "AmCharts.ready()" que é utilizado para sinalizar o fim de abertura da página WEB.

Posteriormente são definidas variáveis que contêm os parâmetros enviados pelo formulário WEB (“id” de utilizador, data de inicio e data de fim). Depois é chamada a função “loadJSON” que irá recolher os dados pretendidos da base de dados. Esta função recebe os parâmetros do formulário e o *url* do outro *script* PHP responsável pela aquisição dos dados (data.php).

A Figura 74 ilustra o processo de aquisição de dados da base de dados.

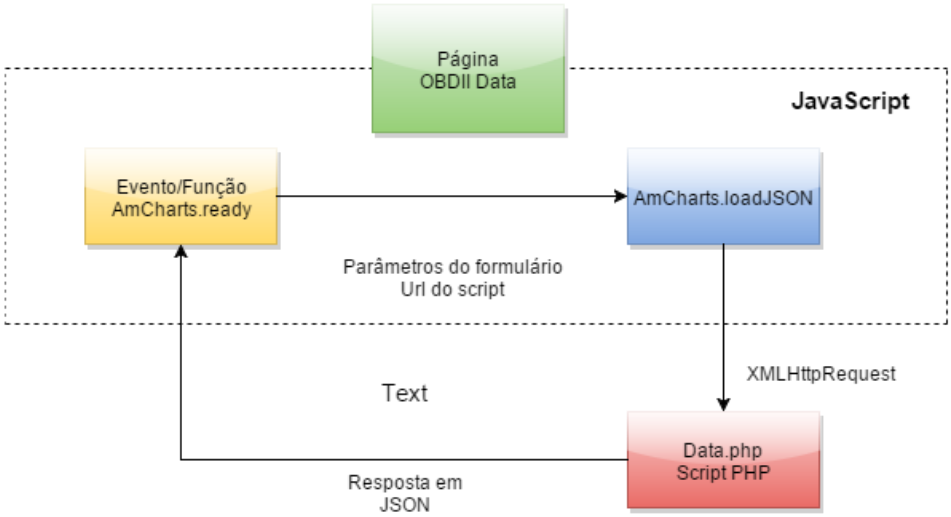


Figura 74 – Processo de aquisição de dados da base de dados

O *script Javascript* contido na página WEB utiliza novamente, e tal como nos *scripts* PHP reponsáveis pela aquisição de dados para as páginas com mapas, um objecto “XMLHttpRequest” para obtenção dos dados armazenados na base de dados.

No que diz respeito ao *script* PHP “data.php”, este simplesmente recebe os parâmetros “id”, “date_begin” e “date_end” e faz posteriormente o pedido SQL à base de dados [48]. Após a aquisição de dados ter sido concluida, o *script* (JavaScript) contido na página WEB, procede à construção do gráfico. A primeira etapa passa por definir, um novo objeto “AmSerialChart”, que expecifica o tipo de gráfico a gerar, definir o “dataProvider” (fonte dos dados) e quais os dados destinados ao eixo das abcissas e o seu formato.

De seguida é adicionado um *cursor* ao gráfico de modo a possibilitar a interação do rato com o gráfico (para movimentar o gráfico por exemplo).

A próxima etapa passa por definir o eixo das abcissas. Este irá conter a data/hora de cada dado OBDII. Uma das vantagens desta biblioteca é o facto que ser possível fazer a formatação do eixo automaticamente para o intervalo temporal apresentado.

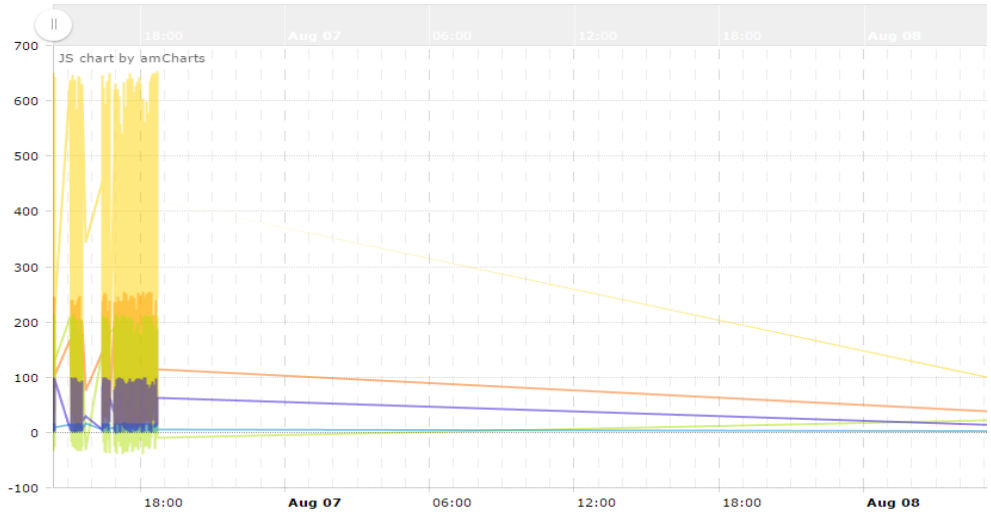


Figura 75 – Gráfico com os dados OBDII de um veículo (gerado na página "OBD Data")

A Figura 75 ilustra o gráfico com os dados OBDII adquiridos para um determinado intervalo de tempo. A formatação do eixo das abcissas ajusta-se automaticamente dividindo neste caso, um dia em quatro parcelas.

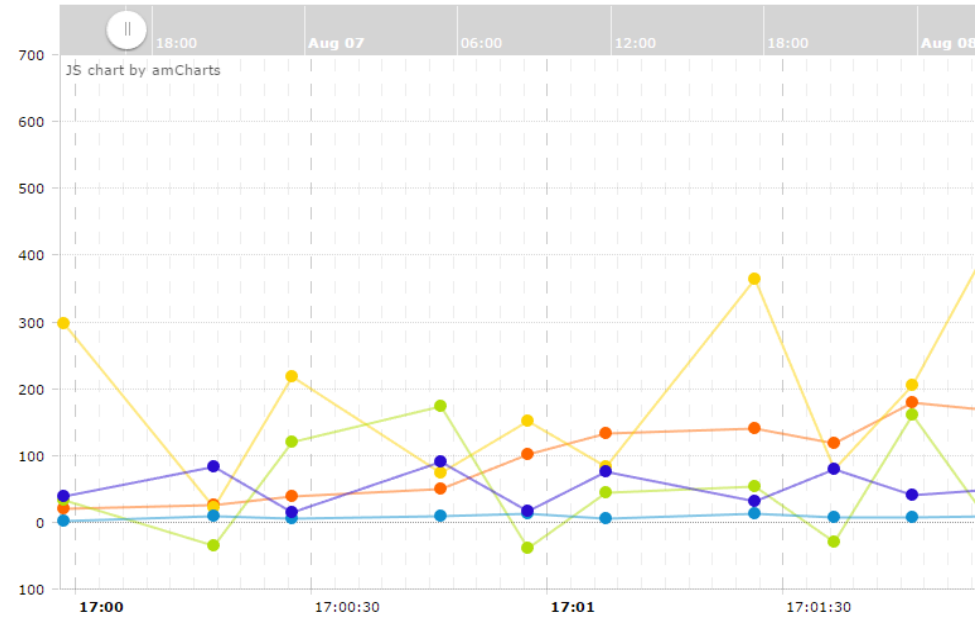


Figura 76 – Gráfico com os dados OBDII de um veículo (gerado na página "OBD Data") em maior pormenor

Ampliando uma determinada parte dos dados o eixo das abcissas é automaticamente formatado mostrando agora a data/hora dos eventos em maior detalhe (consoante a ampliação efetuada), Figura 76. A próxima fase é adicionar os gráficos correspondentes às variáveis OBDII disponíveis na base de dados. Neste caso estão disponíveis cinco variáveis que irão gerar cinco sub-gráficos distintos.

De forma a criar cada sub-gráfico estes têm de ser adicionados individualmente, sendo representados pela classe AmCharts.AmGraph. Posteriormente define-se a fonte de dados (elemento da resposta JSON, por exemplo "speedOBD"), título do sub-gráfico e adiciona-se o gráfico. Por fim o sub-gráfico é adicionado ao gráfico geral na "div" "chartdiv" da página WEB, recorrendo ao método "write". O resultado deste processo é ilustrado na Figura 75 e Figura 76 que revela um gráfico interativo e que permite uma análise facilitada ao utilizador.

E. Página "sensor data"

A página "sensor data" disponibiliza ao utilizador uma tabela com os dados recolhidos de sensores Bluetooth na aplicação *Android* de localização, Figura 77.

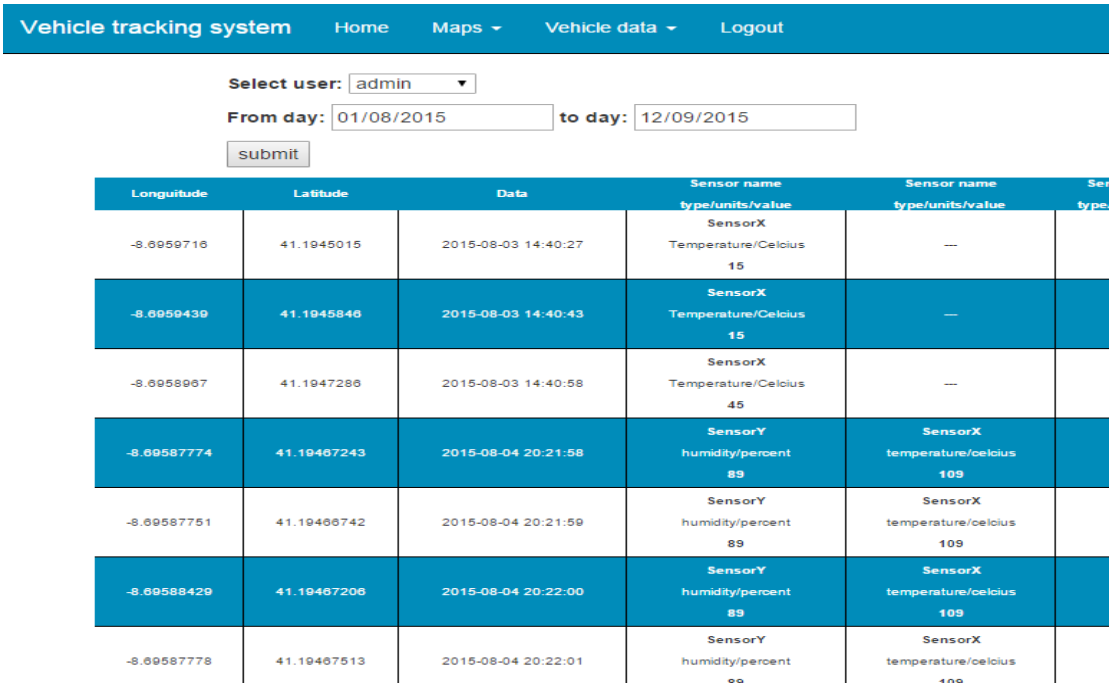


Figura 77 – Página "sensor data"

Esta página é composta por um formulário HTML e uma tabela. Dado que os valores dos dados provenientes dos sensores não têm de ser obrigatoriamente valores numéricos, a criação de um gráfico é largamente dificultada, ou mesmo impraticável. Assim com recurso a uma simples tabela é possível apresentar todo o tipo de dados ao utilizador.

A implementação desta página é bastante similar à das páginas "Latest Vehicles Location" e "Vehicle location history". A diferença mais significativa reside na função "xmlprocessing" que recebe os dados provenientes da base de dados.

A Figura 78 apresenta um exemplo dos dados provenientes da base de dados (XML). Assim a função começa por obter a resposta do servidor e adquirir todos os nós "location", bem como a tabela criada para alojar os dados dos sensores na página WEB.

Figura 78 – Dados adquiridos pelo script PHP

Posteriormente são adquiridos todos os elementos dos nós "location", "locationdata", "sensors" e "sensor".

Após obter os atributos do nó "locationdata" as três primeiras colunas da tabela são preenchidas recorrendo à propriedade "innerHTML" do objeto que representa o elemento HTML. Esta propriedade permite ler ou escrever um determinado elemento HTML, neste

caso um tabela. O próximo passo passa por obter todos os atributos de cada um dos nós "sensor" disponíveis. Por fim é definido o código HTML com que cada célula da tabela será preenchida, contendo os valores dos sensores. A escrita na tabela é novamente efetuada com recurso à propriedade "innerHTML".

5. ANÁLISE DO SISTEMA DESENVOLVIDO

Este capítulo pretende analisar o sistema desenvolvido e verificar se o seu comportamento corresponde ao estipulado.

5.1. EQUIPAMENTO UTILIZADO

Com o intuito de testar o sistema desenvolvido no terreno, a aplicação de localização foi instalada num dispositivo *Android* real (*Samsung Galaxy Ace*). De forma a testar a ligação e comunicação com múltiplos sensores Bluetooth foram utilizados outros *smartphones Android*, utilizando uma aplicação *BlueSPP* [49], disponível gratuitamente na *PlayStore* do *Android*. Esta aplicação suporta comunicação segundo o perfil SPP do *Bluetooth*, permitindo assim simular totalmente uma possível rede de sensores *Bluetooth* no sistema. Deste modo a implementação física do dispositivo sensor *Bluetooth* desenvolvido não foi necessária para o teste do sistema, dado que as mensagens enviadas pelo microcontrolador podem ser perfeitamente simuladas pela aplicação *BlueSPP*.

Para o teste da comunicação com um veículo foi utilizado um adaptador OBDII/*Bluetooth* baseado no microcontrolador ELM327, ilustrado na Figura 79.

Na fase de desenvolvimento, bem como na análise de resultados foi também utilizado o programa de simulação OBDII "ObdSim" [53] no sistema operativo *Microsoft Windows* o que permite analisar o desempenho e funcionamento da aplicação desenvolvida.

Figura 79 – Adaptador OBDII/*Bluetooth* utilizado

5.2. ANÁLISE DO FUNCIONAMENTO DO SISTEMA DESENVOLVIDO

Após a instalação da aplicação num dispositivo *Android*, foram *efetuados* inúmeros testes com um veículo real (Audi A4 de 2004 - protocolo OBDII: iso9141-2) de modo a verificar o correto funcionamento do sistema.

A aplicação funciona exatamente como esperado, registando a localização do veículo a cada período de tempo definido pelo utilizador ou distancia percorridas pelo veículo. Verificou-se que o sistema efetivamente funciona em dois modos distintos, *data-pusher* e *data logger*, consoante o estado da ligação à Internet.

O sistema foi testado diversas vezes e a diferentes velocidades do veículo, correspondendo sempre da forma prevista. A verificação destes resultados obtidos pela aplicação de localização, foi efetuada pela consulta da tabela "*locations"* na base de dados Mysql diretamente, e em tempo real, bem como através da leitura da base de dados interna da aplicação *Android*.

A Figura 80 apresenta os dados OBDII obtidos num percurso efetuado por um veículo real (visualizados na aplicação WEB desenvolvida). O gráfico é interpretado da seguinte forma:

- Linha de cor laranja - Velocidade do veículo (km/h);
- Linha de cor verde - Temperatura do líquido de refrigeração do motor (ºC);
- Linha de cor azul-escuro - Posição do acelerador (%);
- Linha de cor ciano – Rotações do motor (x1000);
- Linha de cor laranja – Fluxo de massa de ar (g/s).

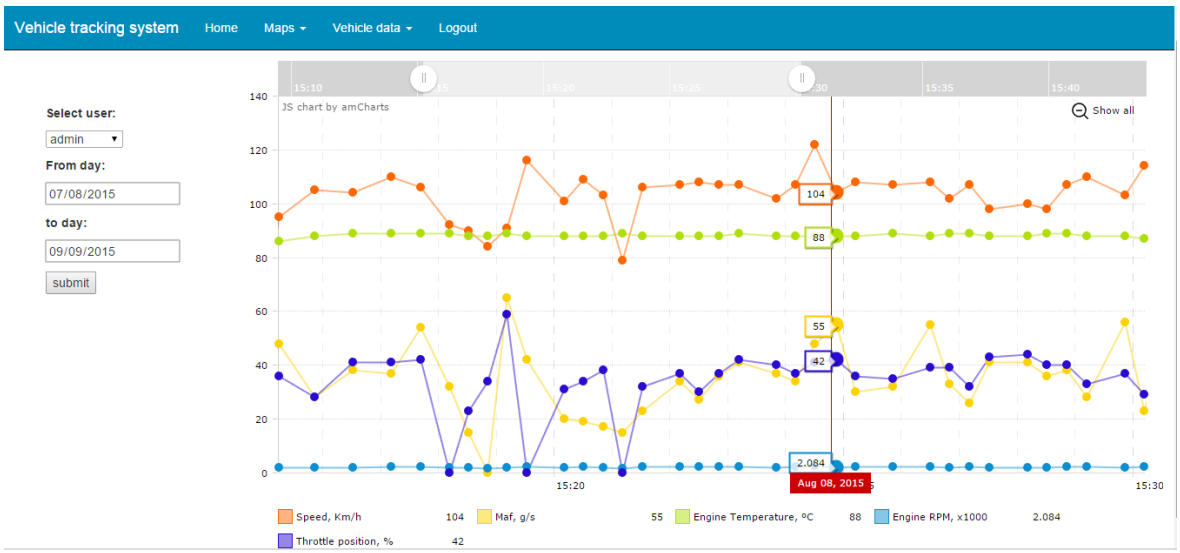


Figura 80 – Dados OBDII obtidos (visualizados na aplicação WEB desenvolvida)

A Figura 81 ilustra a janela da linha de comandos do *Microsoft Windows* com o programa *OBDSim* em execução.

Os parâmetros utilizados para a execução do programa são os seguintes:

- "Obdsim.exe" – Simulador OBDII/ELM327
- "-g guifltk" – Gerador dos valores OBDII (neste caso é definido o interface gráfico)
- "-w COM4" - Porta COM utilizada pelo *Bluetooth*
- "-q LOG.txt" – Registo de comunicações no ficheiro LOG.txt

Na Figura 81 é verificado que o simulador recebeu e executou os comandos utilizados na inicialização do ELM327.

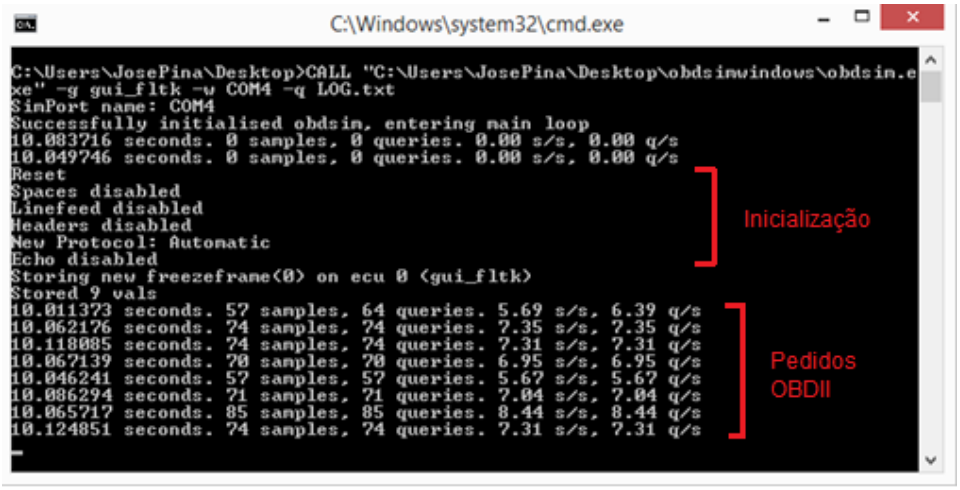


Figura 81 – Simulador OBDSim em execução

Posteriormente é possível visualizar as estatísticas da comunicação, como a média de pedidos por segundo, que neste caso varia entre 5.67 e 8.44 pedidos por segundo, estando assim, em linha com a maioria dos programas OBDII baseados no ELM327, como é o caso da aplicação *Torque* para *Android*.

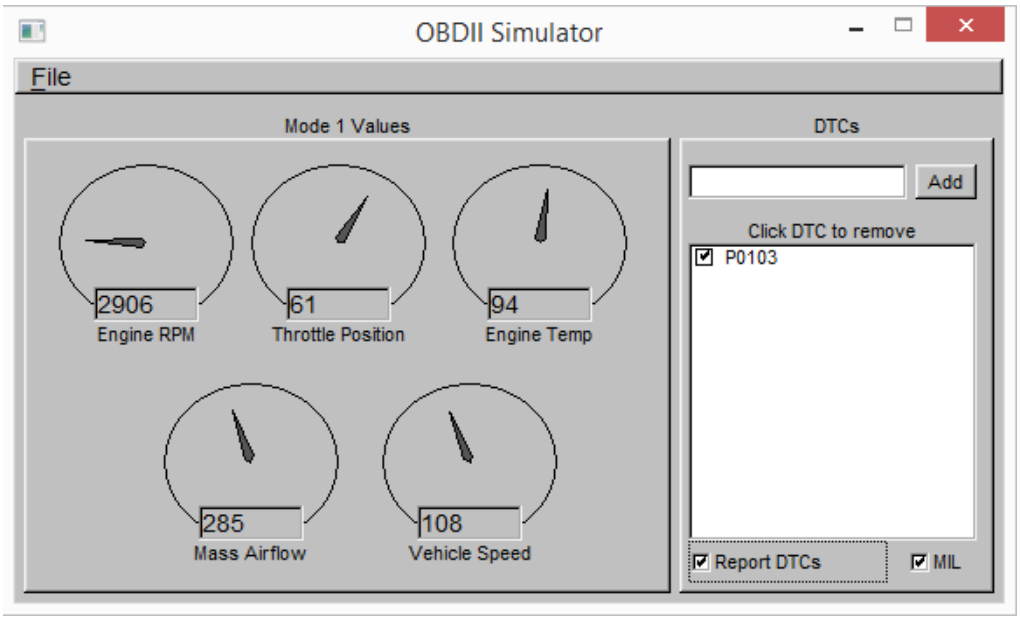


Figura 82 – *Interface* gráfico do simulador OBDSim

Dado que o programa *OBDSim* faz uma simulação do ELM327, o *sample rate* obtido na leitura de dados OBDII deste e do veículo de teste, teve resultados semelhantes. *O sample rate* obtido é diretamente influenciado tanto pelo protocolo OBDII utilizado pelo veículo, bem como pelo *interface* (adaptador OBDII/*Bluetooth*) utilizado.

A Figura 82 ilustra o interface gráfico do simulador *OBDSim*. Do lado esquerdo do *interface* gráfico estão disponíveis cinco mostradores que permitem alterar os respetivos valores reportados pelo simulador a uma aplicação (pedidos no modo 01). Do lado direito podem ser adicionados DTC's para simular a presença de códigos de erro na ECU do veículo.

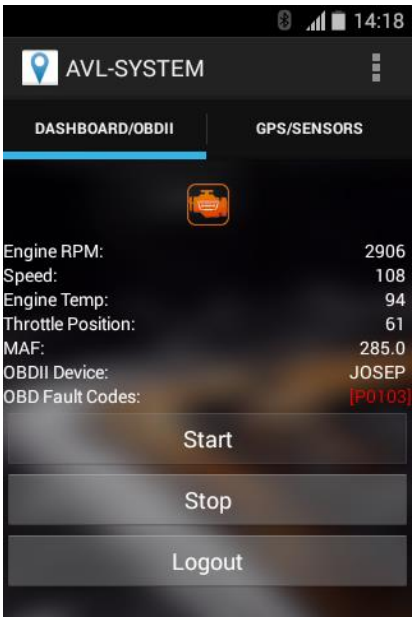


Figura 83 – Dados OBDII do veiculo obtidos na Aplicação Android

A Figura 83 apresenta os valores obtidos do simulador *OBDSim* na aplicação desenvolvida. Assim é verificado que os dados OBDII reportados pela ECU do veículo correspondem ao apresentado na aplicação *Android* e aplicação WEB. No que diz respeito às mensagens enviadas pelos sensores (simulados através da aplicação *BlueSPP*), estas também correspondem ao esperado.

No entanto um pequeno problema foi detetado na comunicação com o ELM327 num caso específico. Se o veículo estiver o motor a trabalhar e a aplicação *Android* ligada, tudo funciona como previsto. Se entretanto o motor do veículo for desligado, não existe perda de ligação *Bluetooth* mas o ELM327 envia a *string* "NO DATA" a qualquer posterior pedido OBDII feito pela aplicação *Android*.

Segundo a *datasheet* do fabricante [26], o ELM327 envia a *string* "NO DATA" nos seguintes casos:

- Sem resposta da ECU do veículo;
- ECU não pode fornecer dados a um PID particular;
- Modo OBDII não suportado;
- ECU do veículo está ocupada com problemas de maior prioridade;
- Resposta ignorada (veículos com protocolo CAN);

Dada a multiplicidade de possíveis interpretações da resposta, não é possível determinar com certeza se a ignição do veículo está ou não desligada.

Após uma análise ao histórico de versões do ELM327 disponível na *datasheet* disponibilizada pelo fabricante, a versão 1.4 do ELM327 introduz um novo comando "AT IGN" que permite determinar o estado da ignição do veículo. No entanto a versão utilizada do ELM327 é anterior à versão 1.4 (apesar de reportar versão 2.1) e não fornece suporte a este comando.

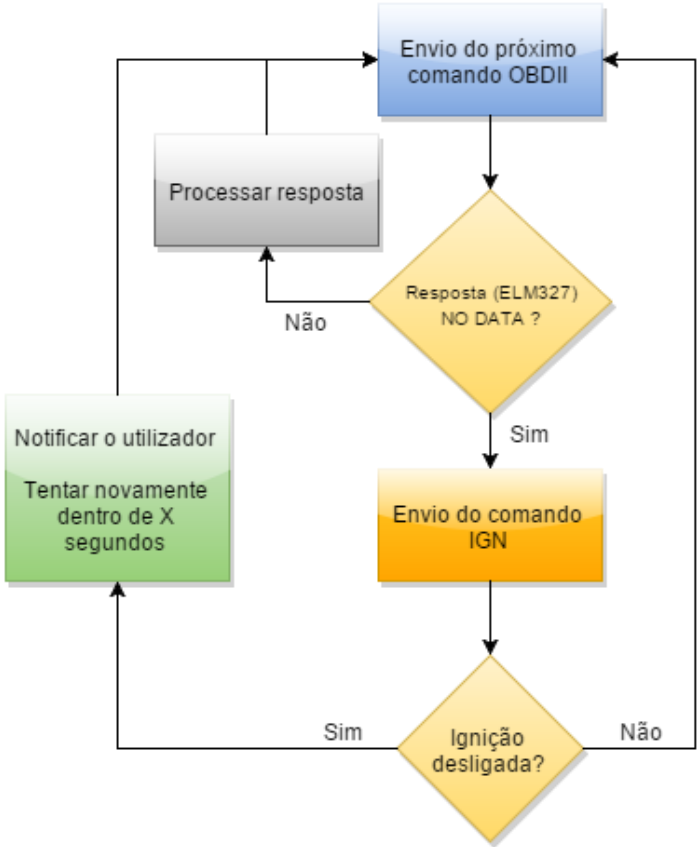


Figura 84 – Algoritmo para verificação do estado da ignição do veículo

Para atender a este problema seria necessário uma verificação adicional na aplicação *Android*. A Figura 84 apresenta um algoritmo desenvolvido que poderia ser utilizado verificar o estado da ignição do veículo através do ELM327 com versão "1.4" e superior.

No caso de a aplicação receber uma resposta do ELM327 contendo a *string* "NO DATA" seria enviado o comando "AT IGN" que reportará o estado da ignição do veículo.

Estando a ignição desligada, a aplicação notifica o utilizador do sucedido e pode, por exemplo, tentar o envio do próximo comando OBDII passado um determinado intervalo de tempo, ou então perguntar ao utilizador se pretende terminar ou não a aplicação. No caso de a ignição do veículo estar ligada, a aplicação poderia enviar o próximo comando OBDII, ignorando o sucedido ou fazer verificações adicionais (se a próxima resposta a um comando OBDII contiver a *string* "NO DATA" novamente).

A fiabilidade do adaptador OBDII/*Bluetooth* utilizado foi outro dos parâmetros analisados. Este adaptador revelou-se pouco fiável no que diz respeito tanto à ligação *Bluetooth* com o *smartphone Android* como com o veículo, o que sugere que um adaptador de maior qualidade (como por exemplo os adaptador da marca Obdlink [54]) deverá ser utilizado de modo a alcançar os níveis de fiabilidade requeridos para um sistema de localização de veículos.

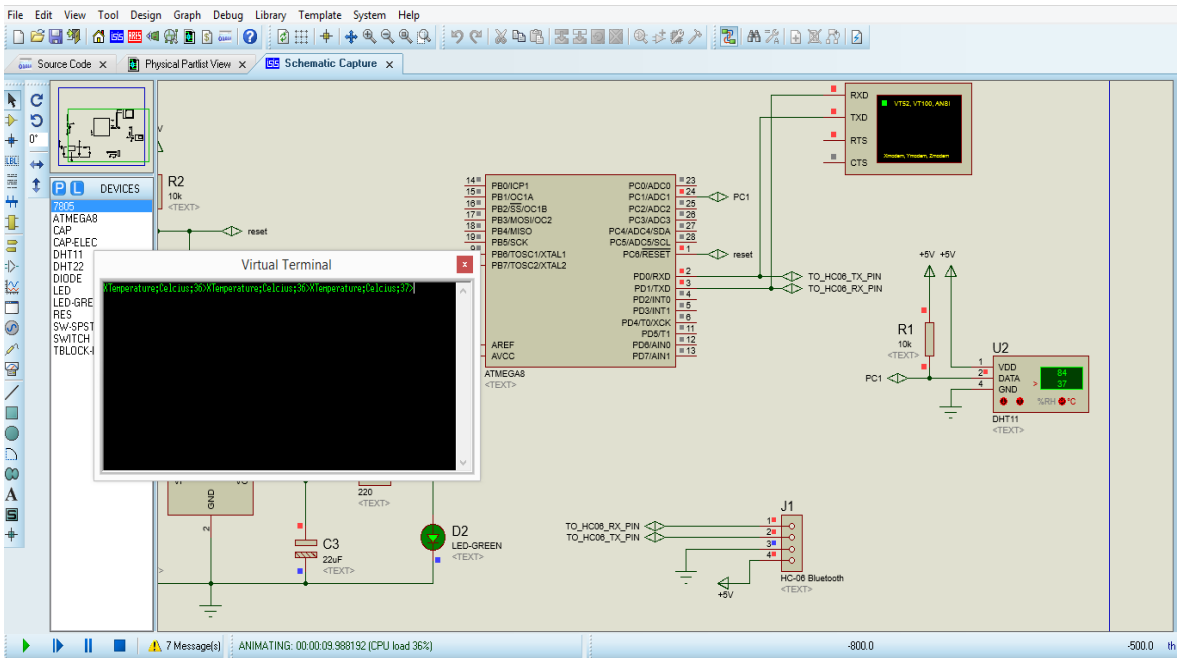


Figura 85 – Programa *Labcenter Electronics Proteus 8*

No que diz respeito ao dispositivo sensor *Bluetooth*, e de modo a testar o dispositivo desenvolvido, foi utilizado o programa *Labcenter Electronics Proteus 8*, Figura 85, que permite a simulação do programa criado para o microcontrolador Atmega8.

O componente *Virtual Terminal* do programa é utilizado para simular a comunicação entre o microcontrolador e a aplicação *Android*. Como visto na secção 4.3.2 a aplicação *Android* envia o caracter "X" para o dispositivo sensor e este responderá, neste caso, com a leitura de temperatura proveniente do sensor DHT-11.

A Figura 86 ilustra a resposta do microcontrolador a um caracter "X" recebido. Tal como esperado, a leitura de temperatura proveniente do sensor DHT-11 será enviada para a aplicação *Android*, o que fará com que esta envie o caracter "X" novamente, pedindo uma nova leitura ao dispositivo.

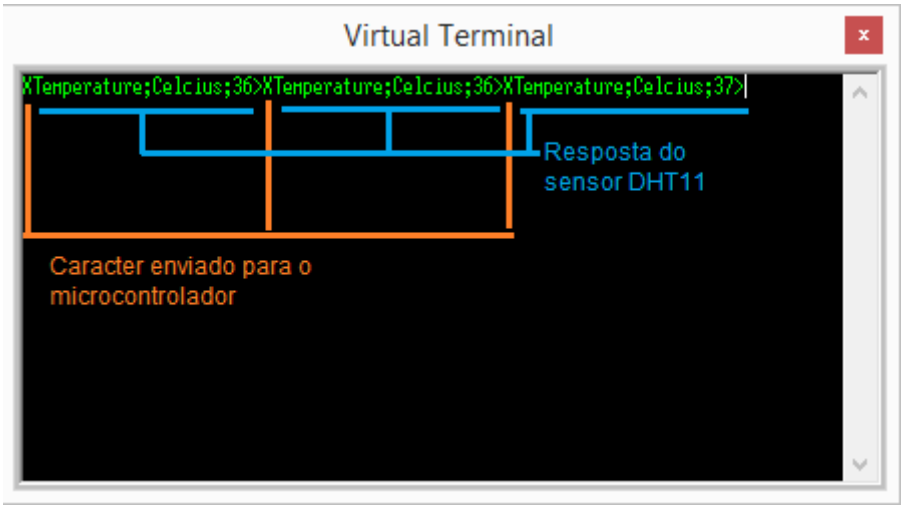


Figura 86 – *Virtual Terminal* do programa *Labcenter Electronics* Proteus 8 – Resposta do microcontrolador a um caracter recebido ("X")

6. CONCLUSÕES

Os *Smartphones* permitem hoje em dia realizar tarefas que no passado estavam unicamente a cargo de equipamentos dedicados. A plataforma *Android* abre um caminho infinito de oportunidades para os utilizadores, surgindo no mercado dispositivos inovadores e com novas funcionalidades quase diariamente. Este trabalho descreve como um sistema de localização de veículos pode ser atualmente implementado com recurso a um simples smartphone. Para além disso o trabalho desenvolvido permitiu um segundo contacto com a plataforma *Android*, que se demonstrou enormemente complexo, mas com muita documentação disponível para um programador. Os objetivos inicialmente propostos foram inteiramente cumpridos e dentro dos prazos estipulados.

Quando ao sistema desenvolvido, este revelou-se completamente funcional, e permitiu analisar e estudar algumas das possibilidades da plataforma *Android*. O sistema apresenta vantagens face a dispositivos dedicados, nomeadamente a facilidade de instalação e portabilidade dos dispositivos, A possibilidade de interação com um adaptador OBDII *Bluetooth* e com até 7 dispositivos *Bluetooth* (sensores ou outros) simultaneamente, torna o sistema extraordinariamente versátil permitindo que este seja utilizado em inúmeras aplicações, mesmo que não em veículos. O seu reduzido custo de aquisição face a um equipamento dedicado é também uma vantagem importante.

Uma possível desvantagem dum sistema baseado em *smartphone* é o facto dos dispositivos dedicados serem especialmente concebidos para garantir uma proteção completa tornando a adulteração uma quase impossibilidade. Mesmo quando escondidos, um *smartphone* não consegue oferecer o mesmo nível de segurança.

O adaptador OBDII/*Bluetooth* utilizado revelou algumas fragilidades ao nível da fiabilidade, o que sugere que um adaptador de maior qualidade deverá ser utilizado.

O sistema poderá no futuro ser melhorado, nomeadamente no aumento de funcionalidades disponíveis. A nível da aplicação poder-se-ia permitir a leitura de mais parâmetros através do sistema OBDII (possivelmente à escolha do utilizador) bem como tornar o *layout* da aplicação um pouco mais apelativo. Poderia também ser introduzido um sistema de mensagens entre o gestor (servidor WEB) e o utilizador.

Referências Documentais

[1] Fleetistics, *Types of GPS Fleet Tracking* - [Consult. 20 Out. 2014] — http://www.fleetistics.com/fleet-tracking-types.php

[2] Wizedrive, *Fleet Tracking Technology* - [Consult. 21 Out. 2014] - http://www.wizedrive.com/

[3] GPS.GOV, *Public information about the U.S. Global Positioning System* (GPS) - [Consult. 23 Out. 2014] - *http://www.gps.gov/systems/gps/*

[4] CORREIA, Carlos- Sistema GPS – introdução, Sistemas de Telecomunicações II. FEUP, 2013 - [Consult. 23 Out. 2014] - http://paginas.fe.up.pt/~hmiranda/st2/galileu.pdf

[5] COOKSEY, Diana - *UNDERSTANDING THE GLOBAL POSITIONING SYSTEM* (GPS), *Montana State University-Bozeman* - [Consult. 24 Out. 2014] - http://www.montana.edu/gps/understd.html

[6] How technology work - *How does a GPS work*- [Consult. 24 Out. 2014] - http://www.howtechnologywork.com/how-does-a-gps-work/

[7] GSM: *Global System for mobile communications* - [Consult. 27 Out. 2014] http://www.4gamericas.org/index.cfm?fuseaction=page&sectionid=242

[8] TK103 GPS Tracker – *Wiring diagram* - [Consult. 28 Out. 2014] https://en.wikibooks.org/wiki/TK103_GPS_Tracker

[9] Trackjinn - *Architecture of Hardware Devices* - [Consult. 28 Out. 2014] - http://www.trackjinn.com/technology.htm

[10] TomTom - *Requirements to use FMS with the LINK 510/530* - https://us.support.business.tomtom.com/app/answers/detail/a_id/3304/~/requirements-to-use-fms-with-the-link-510%2F530

[11] gSat Smart Connections – *Features* - [Consult. 28 Out. 2014] - http://www.gps-server.net/features

[12] MyCarTracks, *Vehicle tracking* - [Consult. 28 Out. 2014] - http://www.mycartracks.com/

[13] INVENTURE—*FMS GATEWAY, CAN INTERFACE FOR TELEMATICS SYSTEMS* - [Consult. 11 Fev. 2015] - http://www.inventure.hu/upload/downloads/INVFMSGWLFEN05.pdf

[14] SIQUEIRA, Thiago - *Bluetooth* – Características, protocolos e funcionamento - [Consult. 6 Fev. 2015] - http://www.ic.unicamp.br/~ducatte/mo401/1s2006/T2/057642-T.pdf

[15] ALECRIM, Emerson - Infowester - Tecnologia *Bluetooth*: o que é e como funciona? - [Consult. 6 Fev. 2015] - http://www.infowester.com/bluetooth.php

[16] MAIA, Eduardo - Redes e protocolos *Bluetooth* - [Consult. 8 Fev. 2015] http://www.academia.edu/1339122/Desvendando_o_bluetooth

[17] DIAS, Ruben - Desenvolvimento de um pedómetro para telemóvel – Dissertação de mestrado, Isep, 2010 - [Consult. 15 Fev. 2015] http://recipp.ipp.pt/bitstream/10400.22/2679/1/DM_RubenDias_2010_MEEC.pdf

[18] Open Security Research - *Bluetooth Device Address* - [Consult. 6 Fev. 2015] http://bnap.opensecurityresearch.com/readme.html

[19] SparkFun - *Bluetooth Basics - Common Versions* - [Consult. 8 Fev. 2015] https://learn.sparkfun.com/tutorials/bluetooth-basics/common-versions

[20] GANESAN, Subra – *Workshop presentation – On Board Diagnostic* (OBDII) for light medium duty vehicles, Oakland University - [Consult. 5 Fev. 2015] http://groups.engin.umd.umich.edu/vi/w2_workshops/OBD_ganesan_w2.pdf

[21] MACHADO, António; OLIVEIRA, Bruno; - O Sistema OBD (*On-Board Diagnosis*) - [Consult. 8 Fev. 2015] - http://ave.dee.isep.ipp.pt/~mjf/act_lect/SIAUT/Trabalhos%202007-08/Trabalhos/SIAUT_OBD.pdf

[22] OBD - *Engine Basics* - [Consult. 6 Fev. 2015] - http://www.onboarddiagnostics.com/page02.htm

[23] OBD - *OBD-II Network Standards* - [Consult. 6 Fev. 2015] - http://www.onboarddiagnostics.com/page03.htm

[24] MCCORD, Keith - CarTech Inc, 2011 - *Automotive Diagnostic Systems* - [Consult. 11 Fev. 2015] - https://books.google.pt/books/about/Automotive_Diagnostic_Systems.html?id=kyEtsrPk9ZQC&redir_esc=y

[25] Ross-tech - *Diagnostic Software for VW-Audi Group Cars* - [Consult. 12 Fev. 2015] http://www.ross-tech.com/vag-com/

[26] Elm Electronics - ELM327 - OBD to RS232 *Interpreter datasheet* - [Consult. 15 Fev. 2015] - http://elmelectronics.com/DSheets/ELM327DSF.pdf

[27] Society of Automotive Engineers - SAE J1979-2002. E/E *Diagnostic Test Modes* - [Consult. 3 Mar. 2015]

[28] Society of Automotive Engineers - SAE J2012-2002.*Diagnostic trouble code* definitions - [Consult. 3 Mar. 2015]

[29] OHA – *Open Handset Alliance - Open mobile platform* (Android) - http://www.openhandsetalliance.com/

[30] Edureka - *The Beginner's guide to Android: Android Architecture,* 2013 – [Consult. 07 Out. 2014] - http://www.edureka.co/blog/beginners-guide-android-architecture/

[31] Google – *Android Developers Guide* - [Consult. 08 Out. 2014] - https://developer.android.com/guide/index.html

[32] Google – *Developers guide - Location Strategies* - [Consult. 10 Out. 2014] http://developer.android.com/guide/topics/location/strategies.html

[33] Google – *Developers guide – Bluetooth* - [Consult. 10 Fev. 2015] http://developer.android.com/guide/topics/connectivity/bluetooth.html

[34] Oracle - *MySQL Tutorial - MySQL 5.1 Reference Manual*, 2014 - [Consult. 20 Out. 2014] - http://www.tutorialspoint.com/mysql/mysql-introduction.htm

[35] TAMADA, Ravi - *Android Login and Registration with PHP, MySQL and SQLite* – 2012, [Consult. 20 Out. 2015] - http://www.androidhive.info/2012/01/android-login-and-registration-with-php-mysql-and-sqlite/

[36] Google – *Google Maps Javascript API - Developers guide tutorial* - [Consult. 5 Abril. 2015] - https://developers.google.com/maps/documentation/javascript/tutorial

[37] Google - *Android Developers Guide – AsyncTask* - [Consult. 28 Out. 2014] http://developer.android.com/reference/android/os/AsyncTask.html

[38] Google - *Android Developers Guide – Settings* - [Consult. 2 Maio 2015] - http://developer.android.com/guide/topics/ui/settings.html

[39] Google – *Android Developers Reference – Handler* - [Consult. 17 Mar. 2015]- http://developer.android.com/reference/android/os/Handler.html

[40] *Stackoverflow* forum – *How to check is a service is running in Android* - [Consult. 25 Out. 2014] - http://stackoverflow.com/questions/600207/how-to-check-if-a-service-is-running-in-android

[41] *Stackoverflow* forum - *Split a string, at every nth position* - [Consult. 27 Fev. 2015] - http://stackoverflow.com/questions/12295711/split-a-string-at-every-nth-position

[42] Google – *Android Developers Reference – Semaphore* [Consult. 28 Junho. 2015] - http://developer.android.com/reference/java/util/concurrent/Semaphore.html

[43] Google – *Android Developers Reference – SQLiteCursor* - [Consult. 27 Jun. 2015] - http://developer.android.com/reference/android/database/sqlite/SQLiteCursor.html

[44] Google – *Google Maps API - Using PHP/MySQL with Google Maps* - [Consult. 1 Jun. 2015] - https://developers.google.com/maps/articles/phpsqlajax_v3

[45] W3schools - XML DOM Tutorial - [Consult. 3 Jun. 2015] - http://www.w3schools.com/xml/dom_intro.asp

[46] IBM – Developerworks - Lendo e Escrevendo DOM XML com PHP - [Consult. 3 Jun. 2015] - https://www.ibm.com/developerworks/br/library/os-xmldomphp/

[47] AmCharts - JavaScript / HTML5 charts - [Consult. 10 Aug. 2015] - http://www.amcharts.com/

[48] AmCharts - *Using PHP to hook up charts to MySQL data base* - [Consult. 10 Aug. 2015] - http://www.amcharts.com/tutorials/using-php-to-hook-up-charts-to-mysql-data-base/

[49] BlueSPP – *Android App* - [Consult. 23 Mar. 2015] https://play.google.com/store/apps/details?id=com.shenyaocn.android.BlueSPP&hl=pt_PT

[50] ATMEL – AVR Atmega8 *datasheet* - [Consult. 25 Set. 2015] http://www.atmel.com/Images/Atmel-2486-8-bit-AVR-microcontroller-ATmega8_L_datasheet.pdf

[51] Micropik – *DHT*-11 *sensor datasheet* - [Consult. 27 Set. 2015] - http://www.micropik.com/PDF/dht11.pdf

[52] Guangzhou HC Information Technology Co., Ltd. - HC-06 *datasheet* - [Consult. 25 Set. 2015] - https://www.olimex.com/Products/Components/RF/BLUETOOTH-SERIAL-HC-06/resources/hc06.pdf

[53] BRIGGS, Gary - OBDSim - [Consult. 18 Fev. 2015] - http://icculus.org/obdgpslogger/obdsim.html

[54] Obdlink - *Scan tools* - http://www.obdlink.com/