

**UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA**

Bruno Luiz da Silva

**ESTUDO E IMPLEMENTAÇÃO DE ESTAÇÃO DE
CARREGAMENTO DE VEÍCULOS ELÉTRICOS**

Florianópolis

2017

Bruno Luiz da Silva

**ESTUDO E IMPLEMENTAÇÃO DE ESTAÇÃO DE
CARREGAMENTO DE VEÍCULOS ELÉTRICOS**

Monografia submetida ao Programa
de Graduação em Engenharia Elétrica
para a obtenção do Grau de Enge-
nheiro Eletricista.

Orientador: Prof. Dr. Eduardo Au-
gusto Bezerra

Florianópolis

2017

Ficha de identificação da obra elaborada pelo autor através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

da Silva, Bruno Luiz

Estudo e Implementação de estação de carregamento de veículos elétricos. / Bruno Luiz da Silva;
Orientador, Eduardo Augusto Bezerra -
Florianópolis, SC, 2017.

79 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro Tecnológico.
Graduação em Engenharia Elétrica.

Inclui referências

1. Engenharia Elétrica. 2. Veículos Elétricos 3. EVSE
4. Microcontroladores. I. Augusto Bezerra, Eduardo.
- II. Universidade Federal de Santa Catarina. Graduação em Engenharia Elétrica. III. Estudo e Implementação de estação de carregamento de veículos elétricos

Bruno Luiz da Silva

ESTUDO E IMPLEMENTAÇÃO DE ESTAÇÃO DE CARREGAMENTO DE VEÍCULOS ELÉTRICOS

Esta Monografia foi julgada aprovada para a obtenção do Título de “Engenheiro Eletricista”, e aprovada em sua forma final pelo Programa de Graduação em Engenharia Elétrica.

Florianópolis, 07 de Março 2017.

Prof. Renato Lucas Pacheco, Dr. Eng.
Coordenador do Curso

Banca Examinadora:

Prof. Dr. Eduardo Augusto Bezerra
Universidade Federal de Santa Catarina

Dr. Cesare Quinteiro Pica
Fundação CERTI

Leonardo Kessler Slongo
Universidade Federal de Santa Catarina

When Henry Ford made cheap, reliable cars people said, 'Nah, what's wrong with a horse?' That was a huge bet he made, and it worked

Elon Musk

Resumo

Este trabalho de conclusão de curso tem como objetivo o estudo de estações de carregamento de veículos elétricos (EVSE), com foco em seu *software*. A implementação utiliza o protocolo OCPP que, ao que tudo indica, irá se tornar o padrão oficial para EVSEs, assim como MODBUS para integrar os diversos dispositivos utilizados no projeto. A estação em estudo segue o padrão Europeu e possibilita carregamentos modo 2 e modo 3 (*Mennekes*).

Palavras-chave: Engenharia Elétrica, Veículos Elétricos, Eletroposto, EVSE, Comunicação, Sistemas Embarcados

Abstract

This Final Year Project aims the study of EVSEs (Electric Vehicle Supply Equipment), focusing on its software. It implements the OCPP protocol, which may become standard for EVSEs, as it implements MODBUS to integrate all devices used in the project. The station follows the European standard and enables mode 2 and 3 charging.

Keywords: electronics, EVSE, electric vehicles, charging stations, embedded systems

Lista de Figuras

Figura 1	Exemplo de sequenciamento para carregamentos CA	27
Figura 2	Conektor Tipo 2 - <i>Mennekes</i>	28
Figura 3	Conektor Tipo 2 - CCS Combo	29
Figura 4	Conektor CHAdeMO	30
Figura 5	Conektor Tesla Supercharger	30
Figura 6	BeagleBone Black e OSSO Cape	35
Figura 7	Medidor Kron da série Mult-K	36
Figura 8	Linha de IHMs MT da WEG	37
Figura 9	Leitor RFID LotusSmart	38
Figura 10	Requisição MODBUS do mestre para o escravo	39
Figura 11	Resposta MODBUS do escravo para o mestre	40
Figura 12	Frame de Comunicação MODBUS	41
Figura 13	Organização de mensagem <i>SOAP</i>	43
Figura 14	Diagrama de conexão entre os dispositivos da EVSE	45
Figura 15	Diagrama UML simplificado do projeto	50
Figura 16	Sequência de Inicialização	50
Figura 17	Diagrama de Inicialização de uma Transação pelo usuário	52
Figura 18	Diagrama de Finalização de uma Transação pelo usuário	53
Figura 19	Ferramenta que simula servidor OCPP	56
Figura 20	Disposição dos dispositivos na bancada de testes	57
Figura 21	Disposição dos dispositivos na estação protótipo	58
Figura 22	Estação protótipo final com Mitsubishi Miev estacionado	59
Figura 23	Gráfico Corrente x Tempo para carregamento do BMW i3 utilizando o conector modo 3 <i>Mennekes</i>	60

Lista de Tabelas

Tabela 1	Vendas nos Estados Unidos.....	23
Tabela 2	Function codes / Operações do MODBUS	41
Tabela 3	Operações OCPP suportadas pela EVSE	47

LISTA DE SIGLAS

CA Corrente Alternada

CAN Controller Area Network

CC Corrente Contínua

CLI Command Line Interface

EVSE Electric Vehicle Supply Equipment

EV Electric Vehicle

GPIO General-purpose input/output

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

IHM Interface Humano-Máquina

IO Input/Output - Entrada e Saída

IP Internet Protocol

JRE Java Runtime Environment

MCU Micro Controller Unit

OASIS Organization for the Advancement of Structured Information Standards

OCA Open Charge Alliance

OCPP Open Charge Point Protocol

PWM Pulse Width Modulation

REST Representational state transfer

RFID Radio-frequency Identification

SOAP Simple Object Access Protocol

TCC Trabalho de Conclusão de Curso

TCP/IP Transmission Control Protocol/Internet Protocol

TC Transformador de Corrente

UART Universal asynchronous receiver/transmitter

UML Unified Modeling Language

WSDL Web Services Description Language

XML eXtensible Markup Language

Sumário

1	INTRODUÇÃO	21
1.1	MOTIVAÇÃO	21
1.2	OBJETIVOS	21
1.3	OBJETIVOS ESPECÍFICOS	22
2	REVISÃO BIBLIOGRÁFICA	23
2.1	MERCADO DE VEÍCULOS ELÉTRICOS	23
2.2	MODOS DE CARREGAMENTO	24
2.2.1	Modos Europeus	24
2.2.1.1	Modo 1 - Carregamento Lento	24
2.2.1.2	Modo 2 - Carregamento Lento	24
2.2.1.3	Modo 3 - Carregamento Rápido CA	25
2.2.1.4	Modo 4 - Carregamento Rápido CC	25
2.2.2	Modos Americanos	25
2.2.3	Outros modos	26
2.2.3.1	Indução	26
2.2.3.2	Troca de Baterias	26
2.2.3.3	Tesla Supercharge	26
2.3	SEQUENCIAMENTO ELÉTRICO DE CARREGAMENTOS CA	27
2.4	PADRÕES DE CONECTORES	28
2.4.1	Tipo 2 - Mennekes	28
2.4.2	Tipo 2 - CCS Combo	29
2.4.3	CHAdE MO	29
2.4.4	Tesla Supercharger	30
2.5	PROTOCOLO OCPP	30
3	DESENVOLVIMENTO	35
3.1	DISPOSITIVOS UTILIZADOS	35
3.1.1	BeagleBone Black e OSSO Cape	35
3.1.2	Medidor de Energia Kron Mult-K	36
3.1.3	Phoenix EM-CP-PP-ETH Controller	36
3.1.4	IHM WEG MT	37
3.1.5	Leitor de RFID LotusSmart	38
3.2	MÉTODOS E FERRAMENTAS UTILIZADAS	38
3.2.1	MODBUS	38
3.2.1.1	Funcionamento	39
3.2.1.2	Mapa de Memória	40
3.2.1.3	Envio de informações	41

3.2.2	WSDL e SOAP	42
3.2.3	JUnit e Automação de testes.....	43
3.2.4	Guice e Injeção de Dependências	44
3.3	ESTRUTURA DO PROJETO	44
3.3.1	Especificações da estação	45
3.3.2	Sistema operacional	45
3.3.3	Comunicação entre estação e servidores	46
3.3.4	IHM	47
3.4	SOFTWARE	47
3.4.1	Principais Classes	47
3.4.1.1	AuthHandler	48
3.4.1.2	OcppManager	48
3.4.1.3	ChargePointServer	48
3.4.1.4	ChargePointClient	48
3.4.1.5	HmiManager	48
3.4.1.6	ConnectorSupervisor	49
3.4.1.7	ConnectorManager e Connector	49
3.4.1.8	TransactionManager e Transaction	49
3.4.2	Inicialização	50
3.4.3	Inicialização e Finalização de Transações pelo Usuário	51
4	TESTES E RESULTADOS	55
4.1	FERRAMENTAS DE TESTE	55
4.2	IMPLEMENTAÇÕES E TESTES INICIAIS	56
4.3	TESTES NA ESTAÇÃO PROTÓTIPO	57
5	CONCLUSÃO	65
5.1	RECONHECIMENTO	66
	Referências	67
	APÊNDICE A – Log de Sequenciamento de Carregamento	73

1 INTRODUÇÃO

Desde o início dessa década, as empresas automobilísticas perceberam o grande potencial que o mercado de veículos elétricos possui. Antes, esse mercado era limitado por questões tecnológicas, como durabilidade e carga de baterias, sistemas embarcados e eletrônica de potência. Porém, hoje essas limitações vem sendo superadas, o que re-aqueceu o setor. O carregamento desses veículos é dado por *Electric Vehicle Supply Equipment (EVSE)*.

Uma *EVSE* traz não só desafios na área de eletrônica de potência, mas também na área de sistemas embarcados e *software* em geral, visto que a estação pode ser feita de múltiplos dispositivos que precisam comunicar-se entre si e com servidores na nuvem. Esse projeto explora a área do *software* de uma *EVSE*.

1.1 MOTIVAÇÃO

Existem diversos aspectos que podem ser tratados nesse tipo de projeto. Há, dentro da estação, dispositivos que permitem medir e atuar sobre os carregamentos. Todos precisam ser controlados por um *software*, o que requer do desenvolvedor conhecimentos de comunicação e projeto de *software*. Além disso, a maioria dos sistemas contemporâneos são conectados a servidores externos, e no caso das *EVSE* não é diferente.

Este trabalho acadêmico explora a implementação do *software* que integra os periféricos da *EVSE* e realiza a comunicação entre a estação de carregamento e um sistema central na nuvem. Tais tarefas permitem um aprendizado mais profundo sobre engenharia de *software*, protocolos de comunicação e integração de múltiplos dispositivos.

1.2 OBJETIVOS

Este Trabalho de Conclusão de Curso tem como objetivo apresentar e implementar um protótipo de estação de carregamento elétrica, que permitirá ao usuário iniciar e cancelar carregamentos, assim como visualizar dados do conector: status (carregando, disponível, veículo conectado) e, caso estiver em um carregamento, tempo total e consumo de energia.

A comunicação com o servidor também é implementada, de modo que este possa receber leituras dos sensores e atualizações dos carregamentos, assim como possibilitar o gerenciamento remoto da estação. Essa comunicação permite a inicialização e finalização de carregamentos de modo remoto, o armazenamento de medições dos carregamentos (corrente, tensão, potência e energia), a verificação do estado dos conectores, a configuração de variáveis internas da estação entre outros. No futuro, tais medições poderão ser utilizadas para efetivar cobranças ao usuário, verificar e atuar remotamente em casos de falhas e permitir que usuários iniciem seus carregamentos por meio de aplicativos de celular.

Para isso são utilizadas tanto tecnologias já consolidadas, assim como outras mais recentes. O projeto é implementado em uma placa de desenvolvimento que possibilita acesso remoto, rodando o sistema operacional *Linux*.

1.3 OBJETIVOS ESPECÍFICOS

- Aprimorar técnicas de desenvolvimento de *software* embarcado
- Conhecer mais sobre as tecnologias envolvidas em sistemas de veículos elétricos
- Explorar e implementar protocolos de comunicação de *EVSE*
- Desenvolver e aprender novas habilidades de engenharia de *software*
- Implementar um *software* para controle de uma *EVSE*

2 REVISÃO BIBLIOGRÁFICA

2.1 MERCADO DE VEÍCULOS ELÉTRICOS

O mercado de veículos elétricos, desde o início da década, está aquecido. Nos Estados Unidos, de 2010 à 2014 houve um aumento de 71% no volume de vendas (286.390 mil veículos em números absolutos), como pode ser visto na tabela 1 (BLOCK; HARRISON; BROOKER, 2015). Na China, em 2015, houve um aumento de 4,2 vezes na produção de carros movidos a bateria, e 3,3 vezes quando incluir outras categorias de *Electric Vehicle (EV)* (CAAM, 2016).

Já na União Europeia, de 2013 para 2014 houve um crescimento de 31,5%, com a venda de 22,6 mil veículos. Embora o crescimento aparente ser grande, está longe de ser maioria no mercado: menos de 1% da atual frota é constituída de veículos elétricos e a adoção destes depende muito dos incentivos dados em cada país. No contexto Europeu, porém fora da União Europeia, a Noruega se destaca, com uma fatia de 22,5 % de veículos elétricos nas vendas de 2015. (EAA, 2016)

Embora esse mercado apresente potencial, ainda há uma grande dependência em fatores externos. Exemplo disso são os atuais fabricantes possuírem poucos modelos, sendo em alguns casos somente uma adaptação de um modelo movido a combustão, pois ainda estão testando o mercado e avaliando seus investimentos.

Alguns fatores influenciam a adoção de veículos elétricos, como:

- Regulamentações exigindo os fabricantes automotores a reduzir emissões de CO₂
- Incentivos fiscais e financeiros para a aquisição de veículos e ope-

Tabela 1 – Vendas nos Estados Unidos

	Híbridos	Somente bateria	Anual	Cumulativo
2010	326	19	345	345
2011	7,671	10,064	17,735	18,080
2012	38,584	14,251	52,835	70,915
2013	49,008	47,694	96,702	167,617
2014	55,357	63,416	118,773	286,390

rações relacionadas

- Preços de combustíveis
- Diminuição de custos de baterias e evolução da tecnologia
- Infraestrutura do ecossistema de veículos elétricos (pontos de carregamento e manutenção)

Alguns países Europeus, como a Alemanha, já afirmaram que desejam abolir vendas de veículos a combustão (SCHMITT, 2016). Outros países planejam seguir a mesma linha, o que força os fabricantes a iniciarem pesquisas na área e desenvolver as tecnologias relacionadas a esse mercado.

2.2 MODOS DE CARREGAMENTO

2.2.1 Modos Europeus

Os modos de carregamento europeus seguem o padrão IEC 62196 (IEC, 2014), que apresenta quatro modos de operação.

2.2.1.1 Modo 1 - Carregamento Lento

Utiliza o conector de tomada residenciais, o qual não pode exceder 16 A de corrente e 250 V Corrente Alternada (CA) monofásica ou 480 V CA de tensão trifásica. Normalmente demanda de 6 à 8 horas para carregar completamente o veículo, o que é considerado um carregamento lento. Como é utilizada uma tomada comum, não é necessária uma *EVSE* para fazer o controle desse carregamento, mas somente uma extensão que conecta a tomada ao veículo. Porém, como esse padrão utiliza uma proteção diferencial e requer uma proteção aterrada na residência, muitos países não o adotam, visto que boa parte das residências não possuem aterramento. Nesse modo, o veículo precisa possuir com um inversor interno para a conversão CA-Corrente Contínua (CC).

2.2.1.2 Modo 2 - Carregamento Lento

É necessário um equipamento de controle no conector ou no cabo, sendo que este equipamento fará o sequenciamento elétrico com o veí-

culo (descrito na seção 2.3). Utiliza o mesmo conector do Modo 1 e possui as mesmas limitações elétricas de tensão, porém permite um máximo de até 32 A de corrente. Dentro desse equipamento de controle há também um circuito de proteção, o que permite o uso desse modo em locais não aterrados. É utilizado em alguns locais públicos da Europa e é considerado uma solução de transição nos EUA. Algumas fabricantes disponibilizam um carregador modo 2 para ser utilizado em casa, visto que é somente um cabo com uma caixa de controle. Nesse modo, o veículo precisa possuir com um inversor interno para a conversão CA-CC.

2.2.1.3 Modo 3 - Carregamento Rápido CA

O cabo é conectado à uma *EVSE*, sendo que essa precisa estar habilitada à realizar *Pulse Width Modulation (PWM)* e possuir uma proteção elétrica interna. O carregamento é ajustado de acordo com o sinal recebido via *PWM*, sendo que seu sequenciamento é explicado na seção 2.3. Com o uso de 400 V trifásico com 63 A de corrente (especificações máximas deste modo), é possível carregar certos carros em menos de uma hora. Esse modo está se tornando cada vez mais comum, porém - para obter aproveitamento máximo - o veículo precisa possuir um inversor adequado corrente máxima da especificação, o que geralmente não ocorre e acaba limitando a velocidade do carregamento.

2.2.1.4 Modo 4 - Carregamento Rápido CC

Este modo ultra-rápido permite tensões de 400 V e corrente de 200 A, utilizando CC para tal. Como a estação é a responsável pela conversão CA-CC, esse modo requer que esta possua um inversor, sendo desnecessário o inversor do veículo. Estações que permitem carregamento no modo 4 custam mais caro que estações modo 3, devido aos requisitos de segurança e a necessidade de um bom inversor para atingir potências elevadas. Porém, essas estão se tornando cada vez mais atrativas devido a velocidade de seus carregamentos.

2.2.2 Modos Americanos

- Nível 1: assim como o modo 1 europeu, utiliza um conector residencial (americano) para o carregamento, fornecendo até 120 V

CA ao veículo.

- Nível 2: pode fornecer 240 ou 208 V CA e de 20 à 100 A. Em instalações residenciais, normalmente acaba limitado à 30 A, podendo oferecer até 7,2 kW de potência. Este é o modo mais comum de instalação em residências americanas.

Há ainda modos de corrente contínua, sendo que esses são idênticos ao Modo 4 Europeu.

2.2.3 Outros modos

2.2.3.1 Indução

O veículo pode ser carregado sem precisar estar conectado à uma estação, o que oferece maior segurança e comodidade para o motorista. Há três maneiras de realizar o carregamento (LONGO et al., 2016):

- Estática: veículo é carregado enquanto está estacionado
- Quasi-estática: o veículo é carregado enquanto há pessoas dentro, porém em locais específicos (parado no trânsito, por exemplo)
- Dinamicamente: enquanto o veículo está em movimento, como em uma rodovia

Ainda há muita pesquisa nesse modo, principalmente devido a sua baixa eficiência quando comparado aos modos cabeados.

2.2.3.2 Troca de Baterias

Há a possibilidade de carregamento por troca de baterias, onde o veículo para em uma estação e um sistema automatizado remove a bateria atual do carro e a substitui por uma carregada (MACHINEDESIGN, 2016). Essa opção oferece segurança para o motorista e não sofre do problema de eficiência do carregamento por indução.

2.2.3.3 Tesla Supercharge

Nos modos cabeados, ainda há o modo proprietário da fabricante Tesla, o *Tesla Supercharge*, que permite o carregamento de 50% da

bateria do Tesla S (até 120kWh) em até 30 minutos (TESLA, 2016). Para possibilitar tal carregamento, os carregamentos são realizados em corrente contínua, assim como o Modo 4 Europeu.

2.3 SEQUENCIAMENTO ELÉTRICO DE CARREGAMENTOS CA

Como descrito na seção 2.2.1, os carregamentos modo 2 e 3 requerem um sequenciamento para a inicialização e finalização de seus carregamentos, definidos pela (IEC, 2014). Esse sequenciamento é dado por meio de *PWM* e variações dos níveis de tensão, o que permite realizar todo controle necessário.

Na figura 1, é possível ver um exemplo do sequenciamento de um carregamento gerenciado por dispositivos da Phoenix Contact. No estado A, quando não há um veículo conectado, o conector apresenta uma saída de 12V -12V constante. Assim que é conectado, a tensão cai para 9V -12V e o estado muda para B.

Enquanto for mantido 9V constante (estado B1), a estação ainda não está pronta para o carregamento, porém assim que começar houver um sinal modulado por *PWM* (estado B2), a estação indica que está pronta. A tensão então cai para 6V -12V ou 3V -12V (estado C ou D) e o veículo inicia o carregamento. Pelo *duty cycle* do *PWM*, a estação pode informar ao veículo quanta corrente pode ser disponibilizada, sendo que durante o carregamento esse *duty cycle* pode ser re-ajustado.

Ao término do carregamento, o veículo desativa o *PWM* e a estação volta para o estado B. Após o conector ser removido pelo usuário, a estação vai para o estado A e permanece assim até o próximo carregamento.

Caso houver uma falha em alguma dessas etapas, o nível de tensão pode ir para 0 -12V ou o sinal é interrompido.



Figura 1 – Exemplo de sequenciamento para carregamentos CA

2.4 PADRÕES DE CONECTORES

Diversos tipos de conectores estão disponíveis hoje no mercado. Um dos padrões mais aceitos para o carregamento lento é o Tipo 2 - Mennekes. Já foi submetido para se tornar o padrão oficial desse tipo de carregamento na Europa (MCKINSEY, 2014).

Para carregamentos rápidos porém, existem três conectores que são bastante usados: o CHAdeMO, o CCS Combo e o Tesla Supercharger. Normalmente, as estações fornecem o conector Mennekes e uma ou duas opções de carregamento rápido, o que é similar ao que ocorre em estações para veículos movidos a combustão (gasolina e álcool no mesmo posto).

2.4.1 Tipo 2 - Mennekes

Proposto pela empresa Mennekes, permite carregamentos CA monofásicos/trifásicos e CC (de baixa e média potências), além de ser retro-compatível com conectores Tipo 1, mesmo possuindo 7 pinos (contra 5 do tipo 1). Permite que a corrente flua de forma bi-direcional, o que permite que os EV possam fornecer energia para a estação, previsto no modelo de *Vehicle-to-Grid*. É muito utilizado em carregamentos de modo 2 e modo 3, o que o tornou padrão na Europa (MENNEKES, 2013), porém para o modo 4 outros tipos são necessários.



Figura 2 – Conector Tipo 2 - *Mennekes*

2.4.2 Tipo 2 - CCS Combo

Permite carregamentos rápidos em CC e CA, sendo uma evolução do Tipo 2 - *Mennekes*. O conector possui o mesmo layout de pinos do *Mennekes*, porém é acrescido de dois pinos extras na sua parte inferior para permitir carregamentos CC (LONGO et al., 2016).



Figura 3 – Conector Tipo 2 - CCS Combo

2.4.3 CHAdeMO

Utilizado em carregamentos modo 3 e 4, foi o primeiro conector que possibilitou carregamentos rápidos e CC. Hoje, porém, outros padrões como o CCS Combo (2.4.2) estão oferecendo um suporte satisfatório para carregamentos rápidos (LONGO et al., 2016). Visto a

preocupação da CHAdeMO com segurança e o alto nível de tensão, esse conector possui 10 pinos.



Figura 4 – Conector CHAdeMO

2.4.4 Tesla Supercharger

Fornecido pela *Tesla Motors*, foca em carregamentos rápidos em CC - modo 4 - e empresas como Nissan e BMW já negociam um acordo com a Tesla para a utilização do conector em seus veículos, o que permite que sua frota utilize a infra-estrutura de carregamentos *Tesla Supercharge* (LONGO et al., 2016).



Figura 5 – Conector Tesla Supercharger

2.5 PROTOCOLO OCPP

As estações de carregamento normalmente se comunicam com um servidor central, que pode gerenciar N estações. Para tal tarefa, é necessário um protocolo de comunicação. Embora ainda não exista um padrão oficial, o *Open Charge Point Protocol (OCPP)* é um padrão *de facto* e já existem esforços para o tornar um padrão oficial junto a *Organization for the Advancement of Structured Information Standards (OASIS)* (OCA, 2016).

Mantido e criado pela *Open Charge Alliance (OCA)*, o *OCPP* está presente em mais de 50 países. Na Europa, todas estações comercializadas precisam ser compatíveis com o *OCPP* e, na América, o interesse da indústria aumentou nos últimos anos (KELLY-DETWILER, 2014).

O protocolo prevê um sistema central que recebe dados de N estações. Caso for necessário, o sistema central pode atuar sob *EVSE* específicas com ações como reservar a estação, cancelar algum carregamento ou até desligá-la. Caso a estação perca conectividade, o protocolo prevê um modo de funcionamento autônomo, somente registrando alguns dados para envio posterior (início e finalização de carregamentos) (OCA, 2012).

Hoje existem duas versões: 1.5 e 1.6. Em ambas versões o protocolo funciona como um *web service*, um tipo de serviço *web* que funciona sob o *stack TCP/IP* e pode possuir diferentes tipos de implementações, como *WebSocket* e *Simple Object Access Protocol (SOAP)*. A versão 1.5, por exemplo, pode ser servida por meio de *WebSocket* ou *SOAP*, sendo que o segundo é servido sob protocolo *Hypertext Transfer Protocol (HTTP)*. Já a versão 1.6 também pode ser servida via *Representational state transfer (REST)*.

As versões 1.6 e 2.0 adicionam novos tipos de requisições, porém é a versão 1.5 que define as requisições base, visto que suas requisições também são suportadas nas versões mais novas. As requisições suportadas pela versão 1.5 são apresentadas na lista a seguir (OCA, 2012):

- Enviadas pela estação:

- Authorize: verifica se o usuário está autorizado à usar a estação
- BootNotification: notifica a inicialização da estação e checa se ela pode se conectar ao servidor
- DataTransfer: requisita variáveis e requisições específicas

que não são padronizadas pelo OCPP

- DiagnosticStatusNotification: notifica o estado do envio dos dados de diagnósticos da estação
- FirmwareStatusNotification: notifica o estado da atualização de *firmware*
- Heartbeat: notifica o servidor que a estação ainda está conectada
- MeterValues: envia medições dos carregamentos da estação (corrente/tensão/potência...)
- StartTransaction: notifica a inicialização de um carregamento
- StatusNotification: notifica a mudança de estado de um conector
- StopTransaction: notifica a finalização de um carregamento

- Recebidas pela estação:

- CancelReservation: cancela a reserva do conector atual
- ChangeAvailability: modifica a disponibilidade de um conector
- ChangeConfiguration: modifica variáveis internas da estação
- ClearCache: limpa o cache da estação (como a lista de usuários que foram autorizados)
- DataTransfer: recebe requisições específicas que não são padronizadas pelo OCPP
- GetConfiguration: requisita as variáveis internas da estação
- GetDiagnostics: requisita dados de diagnóstico da estação
- GetLocalListVersion: requisita a versão da *whitelist* que está sendo utilizada na estação
- RemoteStartTransaction: inicializa um carregamento remotamente
- RemoteStopTransaction: finaliza um carregamento remotamente
- ReserveNow: reserva um conector da estação
- Reset: reinicia a estação
- SendLocalList: envia uma nova *whitelist* para a estação

- UnlockConnector: destrava um conector (caso esse possua trava mecânica controlada pela estação)
- UpdateFirmware: atualiza o *firmware* da estação

Os dados obtidos e as requisições permitidas abrem espaço para algumas aplicações. Com os MeterValues, StatusNotification, ChangeAvailability, StartTransaction e StopTransaction é possível obter um relatório dos carregamentos que ocorreram, realizar a cobrança destes, visto que as transações sempre estão associadas a um usuário, e até atuar no desligamento de um conector em caso de falhas. Requisições como ReserveNow, CancelReservation, RemoteStartTransaction e RemoteStopTransaction permitem que donos de veículos elétricos, por meios remotos (exemplo: aplicativos de celular), atuem na estação de modo à iniciar o carregamento do seu veículo ou agendar a estação para uso posterior. As outras requisições são mais focadas na administração e manutenção da estação, o que também é importante visto que nem sempre existirá uma equipe local para gerenciá-la.

Embora o protocolo implemente diversos tipos de requisições e aplicações, os fabricantes não são obrigados a implementar todas. Isso dá flexibilidade no momento do desenvolvimento da estação.

3 DESENVOLVIMENTO

3.1 DISPOSITIVOS UTILIZADOS

3.1.1 BeagleBone Black e OSSO Cape

A BeagleBone Black é uma placa de desenvolvimento *open-source*, desenvolvida pela *Texas Instruments*, do tamanho aproximado de um cartão de crédito. Embora possua um tamanho reduzido, possui os recursos de hardware necessários para viabilizar a execução de distribuições *Linux* como, por exemplo, o Debian (que foi a utilizada nesse projeto).



Figura 6 – BeagleBone Black e OSSO Cape

A BeagleBone permite expansões por meio de *capes*, placas não-oficiais que permitem melhor explorar o uso das saídas e entradas. Nesse projeto, foi utilizada a OSSO Cape, que dispõe de:

- Oito entradas digitais opto-acopladas
- Oito saídas digitais por meio de relés
- Porta RS-485 integrada
- Porta I2C integrada

- Alimentação de fontes externas com tensões entre 5 à 24V

A utilização da cape permite um uso fácil das entradas e saídas digitais, assim como da comunicação RS-485. Caso não fosse utilizada, seria necessário planejar e fabricar uma placa com as proteções e acionamentos que necessários para as entradas e saídas, assim como montar um circuito para converter a saída Universal asynchronous receiver/transmitter (UART) para RS-485.

3.1.2 Medidor de Energia Kron Mult-K

O medidor de energia é essencial para informar tanto ao usuário, quanto ao sistema central, quanta energia foi consumida durante os carregamentos. O dispositivo escolhido foi o Kron Mult-K, pois esse permite a medição da energia consumida e outros dados, todos disponibilizados via MODBUS-RTU (conexão RS-485).



Figura 7 – Medidor Kron da série Mult-K

3.1.3 Phoenix EM-CP-PP-ETH Controller

Os padrões de carregamento CA requerem um sequenciamento elétrico entre carro e estação. O Phoenix EM-CP-PP-ETH permite gerenciar carregamentos AC trifásicos por meio do *PWM* e variação dos

níveis de tensão, o que permite realizar todo controle necessário entre a inicialização e finalização dos carregamentos (PHOENIXCONTACT, 2016). O processo utilizado por esse dispositivo é o mesmo descrito na seção 2.3.

3.1.4 IHM WEG MT

Para o usuário utilizar a EVSE, é necessária uma interface que apresente informações de forma descomplicada e, além disso, seja robusta a intempéries como calor excessivo e chuva. Para tal, foi escolhida a Interface Humano-Máquina (IHM) da WEG, linha MT.

Utilizando o *software EasyBuilder 8000*, é possível definir as telas que serão exibidas para o usuário, assim como os dados que serão exibidos. Os dados e o controle da IHM podem ser realizados por meio de diversos protocolos, dentre eles o MODBUS-TCP, o qual foi escolhido para o projeto.



Figura 8 – Linha de IHMs MT da WEG

3.1.5 Leitor de RFID LotusSmart

De acordo com a especificação do *OCPP*, para iniciar o carregamento na estação é necessária a utilização de cartões *Radio-frequency Identification (RFID)*, sendo necessário a utilização de um leitor compatível. O dispositivo escolhido, *LotusSmart*, permite apenas a leitura de cartões de 13,56 MHz, visto que há uma variedade de frequências de operação.

Os dados da *tag* são armazenados em um *buffer*, semelhante ao *buffer* de teclado de computadores, e enviados serialmente para o sistema, sendo entregues em formato *ASCII*.



Figura 9 – Leitor RFID LotusSmart

3.2 MÉTODOS E FERRAMENTAS UTILIZADAS

3.2.1 MODBUS

Em projetos industriais, é requisito que dispositivos diversos consigam comunicar-se entre si. Para tal, existem dezenas de protocolos que permitem a realização dessa tarefa, sendo o MODBUS um dos mais comuns (MODBUS, 2012). Sua implementação inicial era apenas no modo serial, porém hoje também é possível utilizar o stack TCP/IP e outras implementações menos comuns - UDP, PEMEX, Enron.

3.2.1.1 Funcionamento

O protocolo é baseado no modelo mestre-escravo, onde um dispositivo mestre requisita os dados dos dispositivos escravos.

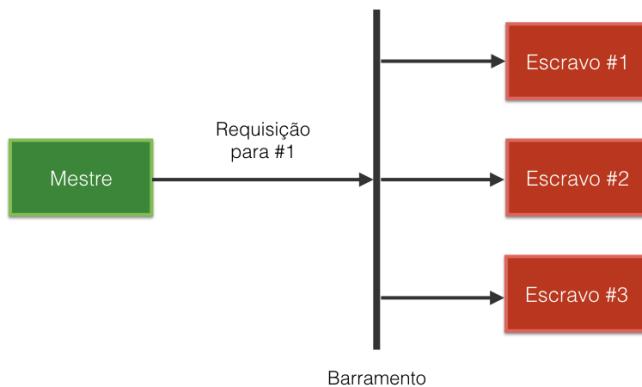


Figura 10 – Requisição MODBUS do mestre para o escravo

Como uma rede pode ter N escravos, cada escravo possui um endereço atribuído. O mestre envia a requisição para o barramento e todos os escravos a recebem, porém somente o endereçado responderá. Há também a opção de se fazer um *broadcast* - endereço 0 - onde todos escravos recebem a requisição, porém nesse caso nenhum escravo deve responde-lá.

Vale notar que em uma rede MODBUS-RTU há somente um mestre, enquanto em uma rede MODBUS-TCP é possível que cada mestre seja um escravo, e vice-versa.

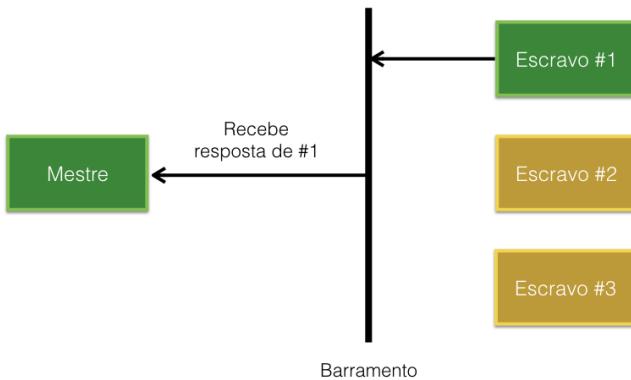


Figura 11 – Resposta MODBUS do escravo para o mestre

3.2.1.2 Mapa de Memória

Toda requisição realizada pelo mestre endereçará um dado no mapa de memória do dispositivo escravo, sendo que tal mapa de memória normalmente deve ser indicado por uma documentação. Os dados requisitados podem possuir quatro tipos distintos:

- *Input*: dado binário que permite apenas leitura
- *Coil*: dado binário que permite leitura e escrita
- *Input Register*: registrador com 16 bits (inteiro) que permite apenas leitura
- *Holding Register*: registrador com 16 bits (inteiro) que permite leitura e escrita

Para a realização de operações nesse mapa de memória, o mestre precisa informar nas suas requisições um *function code*. Este indicará qual dado deseja-se requisitar e se será executada uma operação de leitura ou escrita. A tabela 2 mostra os mais comuns, porém existem mais códigos, aos quais podem ser usados para diagnósticos e outras funções.

Tabela 2 – Function codes / Operações do MODBUS

Operação	Function code
Leitura de Coils	1
Escrita em 1 Coils	5
Escrita em N Coils	15
Leitura de Inputs	2
Leitura de Input Registers	4
Leitura de Holding Registers	3
Escrita em 1 Holding Register	6
Escrita em N Holding Registers	16

3.2.1.3 Envio de informações

Cada modo possui alguma diferença, porém o meio que as informações são organizadas e enviadas é o mesmo. Na figura 12 é possível observar que em ambas implementações é necessária a definição de um *FCode (function code)*. Logo após vem o campo *Data*, onde devem ser indicados o endereço inicial do mapa de memória, quantos registradores deseja-se requisitar e quais dados serão escritos (no caso de escrita).

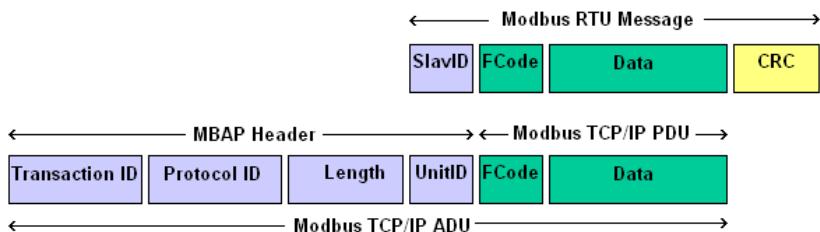


Figura 12 – Frame de Comunicação MODBUS

Como pode-se notar, no MODBUS-RTU há o SlavID, que é utilizado para endereçar qual dispositivo se deseja requisitar (endereço). No MODBUS-TCP há o UnitID, que funciona para a mesma finalidade que o SlavID, e pode ser utilizado em conjunto com o endereço IP do dispositivo. O campo CRC é um dado que permite checar se o pacote obtido possui ou não algum erro. Como o protocolo TCP/IP já possui

um meio para isso, tal campo não é incluído na requisição.

Esse formato de envio de informações é mantido tanto para requisição quanto para resposta, sendo que as respostas sempre possuem o mesmo *function code* da requisição. Caso diferir, provavelmente ocorreu um erro e este pode ser processado pelo mestre posteriormente.

3.2.2 WSDL e SOAP

O WSDL (CHRISTENSEN et al., 2001) permite descrever serviços web por meio de um arquivo *eXtensible Markup Language (XML)*. É possível definir os *endpoints* - pontos de entrada para requisições - com o que devem retornar e o que devem receber como parâmetros. Isso permite que um serviço *web* seja desenvolvido por uma equipe que, posteriormente, exporta um *Web Services Description Language (WSDL)* descrevendo de maneira fácil quais são seus *endpoints*. Isso facilita a criação de clientes ou servidores compatíveis com o serviço criado.

Os arquivos *WSDL* definem suas interfaces em *SOAP* (GUDGIN et al., 2007), um protocolo que permite trocas de informações entre clientes e servidores de forma padronizada, baseada em *XML*. As mensagens *SOAP* são montadas em três partes, que são representadas na figura 13.

- Envelope: identifica o que é a mensagem e como processá-la
- Cabeçalho (*Header*): define informações extras sobre a mensagem
- Corpo (*Body*): define o procedimento a ser chamado e os dados de sua resposta

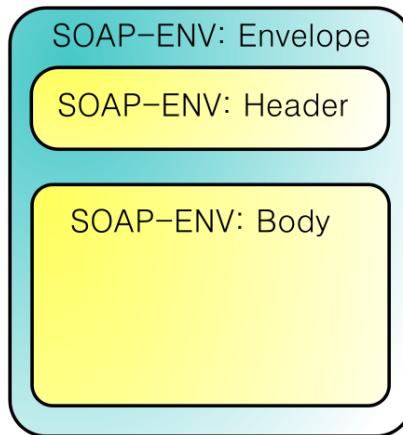


Figura 13 – Organização de mensagem *SOAP*

Como o *WSDL* foi idealizado em *XML*, é facilmente processado por qualquer linguagem de programação, o que garante uma boa portabilidade.

3.2.3 JUnit e Automação de testes

O *JUnit* é um *framework* que facilita a criação de código para a automação de testes unitários. Pode ser verificado se cada método de uma classe funciona da forma esperada, identificando possíveis erros antes de ser enviado para o ambiente de produção (exemplo: servidor ou placa de desenvolvimento) (DEVELOPER.COM, 2004).

Isso é de extrema importância para casos onde a equipe precisa modificar diversas classes, porém precisa manter suas funcionalidades. Assim, caso alguma modificação gerar alguma falha durante os testes, o desenvolvedor rapidamente poderá analisar e corrigir a falha, não deixando esta se propagar no meio de produção, minimizando a chance de *bugs*.

Um dos problemas da utilização de *frameworks* de automação de teste é a exigência de uma boa dedicação para a criação dos testes, algo nem sempre possível devido a limitações de projeto, como prazos de entrega. Outro é a necessidade de uma boa experiência para a criação de testes úteis e assertivos, pois é necessário um balanço entre quanto tempo dedicar e quais funcionalidades são críticas (e devem ser

testadas). Além disso, o ideal seria que um terceiro instrumentasse as classes utilizando o *framework*, e não o desenvolvedor, pois este estará viciado em seu código, enquanto um terceiro procurará por situações não observados pelo desenvolvedor.

3.2.4 Guice e Injeção de Dependências

A injeção de dependências permite que as classes sejam pouco acopladas entre si. Para tal, a lógica de instanciação das dependências não é gerenciada pela classe que as utiliza. O desenvolvedor só necessitará declarar quais dependências (outras classes) deseja e se preocupar em como utilizá-las na classe.

A tarefa de instanciação e injeção dessas dependências é feita por um *service container*, retirando a responsabilidade das instanciações das classes (o que as torna menos acopladas). Existem algumas ferramentas disponíveis que implementam isso de forma eficiente, como o *Guice* e o *Spring* (THESERVERSIDE, 2010). A escolhida foi o *Guice*, desenvolvida e mantida pelo *Google*.

3.3 ESTRUTURA DO PROJETO

A figura 14 apresenta o diagrama de conexão dos dispositivos. Como todos os dispositivos são compatíveis com MODBUS, esse foi o protocolo escolhido para a comunicação entre eles. Para a comunicação entre servidor e estação, foi escolhido o protocolo OCPP 1.5.

O *software* foi desenvolvido em *Java*, visto a facilidade para implementar uma *API SOAP* a partir de um *WSDL*, o conhecimento de outros membros do projeto sobre a linguagem, a quantidade de recursos de suporte na web e a portabilidade fácil do código.

A estação possui dois medidores, sendo um para o conector modo 2 (Med. Kron #1) e outro para o conector *Mennekes* - modo 3 (Med. Kron #2). O controle de cada conector se dá de forma diferente: o de conector *Mennekes* modo 3 utiliza o controlador *Phoenix*, enquanto o conector modo 2 é controlado pela *BeagleBone* e um relé, visto que esse é somente uma tomada comum e não necessita de nada muito elaborado para inicializar carregamentos.

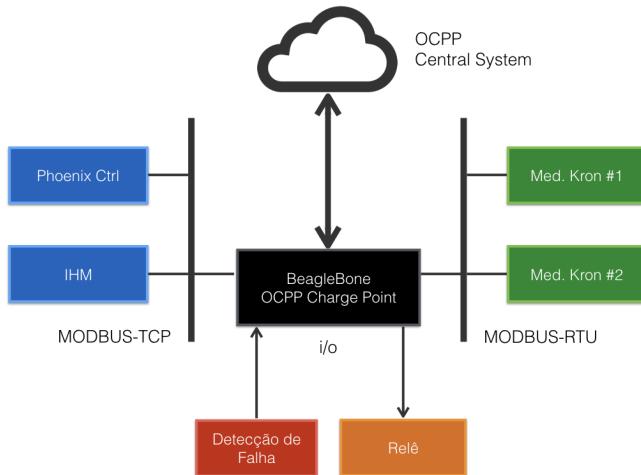


Figura 14 – Diagrama de conexão entre os dispositivos da EVSE

3.3.1 Especificações da estação

O conector *Mennekes* funciona em modo CA trifásico, onde pode fornecer até 32 A sob tensão de 220 V (máxima corrente suportada pelo cabo utilizado). Devido a limitações, como corrente suportada pelo inversor do veículo ou nível de carga da bateria, a corrente consumida pode ser menor do que o limite dado pela estação, como é o caso do BMW i3, que utiliza apenas uma das fases do conector e utiliza somente até 25 A.

O outro conector, é uma tomada padrão ABNT padrão NBR 14136 (ABNT, 2012) que pode fornecer até 20 A. Em carregamentos de carros, o cabo possui uma caixa de controle que se comunica com o veículo e controla a demanda de potência para o carregamento. Em alguns outros veículos, como bicicletas, o cabo possui também um inversor CA-CC, visto que o espaço nesses casos é reduzido.

3.3.2 Sistema operacional

O sistema operacional utilizado é o Debian 8.6 (Jessie) com um kernel *Linux* específico para a *BeagleBone*. Foi instalada a versão mais

recente do *Java Runtime Environment (JRE)*, visto que o projeto utiliza Java, e houveram modificações em sua *device tree*.

A *device tree* é um recurso que permite o usuário configurar as *General-purpose input/output (GPIO)* do sistema para se adequar aos usos do projeto. No caso da *EVSE*, as saídas e entradas são reconfiguradas para que a *OSSO Cape* possa ser utilizada. Além dessa, é necessário ativar outra que habilita a utilização da saída *UART*.

Para utilizar essas *device trees*, é necessário baixar arquivos dts, que descrevem como cada *GPIO* deve ser configurada. Após isso é necessário compilar os arquivos dts e gerar arquivos dtb, tarefa facilmente executada pela ferramenta dtc (inclusa na maioria das distribuições *Linux*).

Após compiladas e instaladas, é necessário ativar essas *device trees* na inicialização. Isso pode ser feito por meio do *uBoot*, o gerenciador de inicialização que vem embarcado na BeagleBone. É necessária a edição do arquivo *uEnv.txt*, normalmente situado na pasta */boot* do sistema, para ativar as *device trees* desejadas.

Foi adicionado também um *script* na pasta */etc/init.d/* para ativar o acesso à internet e executar o programa logo após a inicialização do sistema. Essa pasta é utilizada pelo *System V*, que é um tradicional gerenciador de serviços para o Linux, sendo responsável por inicializar serviços dependendo do nível de inicialização que o sistema se encontra.

O nível 0 representa o desligamento do sistema. O nível 1 representa um modo mínimo, sem conectividade e de usuário único. Já do modo 2 ao 5, o sistema possui conectividade, permite múltiplos usuários e pode possuir *Graphical User Interface (GUI)* (normalmente nível 5). É nesses modos que o *script* de inicialização da *EVSE* é executado. Ainda existe o nível 6, que representa uma reinicialização do sistema.

3.3.3 Comunicação entre estação e servidores

A estação implementa o protocolo *OCPP* 1.5, ao qual é suportado por outras estações comerciais vendidas no Brasil, permitindo comunicação com equipamentos e *software* já existentes. A *OCA*, responsável pelo protocolo, disponibiliza um arquivo WSDL, onde é possível criar tanto um servidor quanto um cliente *OCPP* facilmente.

Para a utilização desse arquivo em *Java*, foi utilizada a ferramenta *wsimport*. Essa ferramenta importa os arquivos e cria classes *Java* correspondentes a cada *endpoint*. Nem todas funções especificadas pelo protocolo foram implementadas, sendo que somente as da

Tabela 3 – Operações OCPP suportadas pela EVSE

Enviada/Recebida	Operação
Enviada pela Estação	BootNotification
Enviada pela Estação	Heartbeat
Enviada pela Estação	StartTransaction
Enviada pela Estação	StopTransaction
Enviada pela Estação	MeterValues
Enviada pela Estação	StatusNotification
Enviada pela Estação	Authorize
Recebida pela Estação	RemoteStartTransaction
Recebida pela Estação	RemoteStopTransaction
Recebida pela Estação	ChangeConfiguration
Recebida pela Estação	GetConfiguration
Recebida pela Estação	ChangeAvailability

tabela 3 são suportadas.

3.3.4 IHM

O software da IHM não foi desenvolvido pelo acadêmico, porém são de importância fundamental para o projeto. A escolha das telas foi realizada em equipe e implementada no software *EasyBuilder 2000*.

A interface é atualizada a cada 200 milissegundos para evitar dados desatualizados ou respostas lentas à entradas do usuário. Ainda assim foram observados atrasos, ligados a capacidade de processamento da IHM com requisições muito frequentes a seus registradores. A comunicação e gerenciamento da IHM se dá pelo *HmiManager*, apresentado na seção 3.4.

3.4 SOFTWARE

3.4.1 Principais Classes

Devido a complexidade do projeto de software e suas diversas classes e implementações, a lista abaixo resume as principais classes que permitem o funcionamento da estação. Essas serão referidas nas seguintes seções sobre o funcionamento da estação.

3.4.1.1 AuthHandler

Gerencia autenticação via *RFID*. Primeiramente verifica se o usuário está habilitado por meio de uma *whitelist* e, caso não encontre nenhum dado nela, faz uma requisição ao servidor *OCPP* para a verificação.

3.4.1.2 OcppManager

Roda em uma instância própria e, a cada período de tempo, executa a lista de comandos que foram adicionados à sua fila (possui um método público *queue()* que permite isso). Implementa o *Dispatcher* do *Command Pattern* (GAMMA et al., 1995), onde os seus comandos são implementações das requisições da tabela 3, e roda em uma instância individual.

3.4.1.3 ChargePointServer

Implementação da interface *WSDL* gerada pelo *wsimport*. Roda em uma instância individual e cria um servidor ao qual o *Central System* pode requisitar dados. Esse servidor permite que o *Central System* modifique configurações internas da estação, inicialize e finalize carregamentos, modifique estados de conectores dentre outros.

3.4.1.4 ChargePointClient

Implementação da interface *WSDL* gerada pelo *wsimport*. Permite enviar requisições para o *Central System* e é chamado apenas por comandos despachados pelo *OcppManager*.

3.4.1.5 HmiManager

Gerencia os estados da IHM, sendo que todos estados são representados por classes. Implementa o *Context* do *State Pattern* (GAMMA et al., 1995), sendo que cada estado (implementações da classe *ScreenState*) pode modificar qual será o estado da próxima iteração por meio de informações adquiridas do dispositivo IHM ou de dados vindos de

outras dependências. Os principais estados são: DefaultScreenState, que mostra ao usuário os dados das *Transactions* atuais e permite a autenticação, e o ConnectorSelectionScreenState, que permite a inicialização ou finalização do carregamento por parte do usuário. Roda em uma instância própria.

3.4.1.6 ConnectorSupervisor

Roda em uma instância, onde checa periodicamente o status dos conectores, envia informações para o servidor sobre dados de medição destes e expira antigas reservas (esse último será implementado em versões futuras).

3.4.1.7 ConnectorManager e Connector

Classe que armazena todos conectores, sendo cada conector um objeto *Connector*. As classes *Connector* gerenciam cada conector da estação, o que possibilita a obtenção de dados e controle por meio de diferentes interfaces (*Input/Output - Entrada e Saída (IO)* e MODBUS no momento).

3.4.1.8 TransactionManager e Transaction

Classe que armazena transações ativas, sendo que transações são instâncias da classe *Transaction*. Cada transação roda em uma instância individual, sendo que estas são responsáveis pelos processos de inicialização e finalização junto ao *OCPP*, assim como envio periódico de dados da transação.

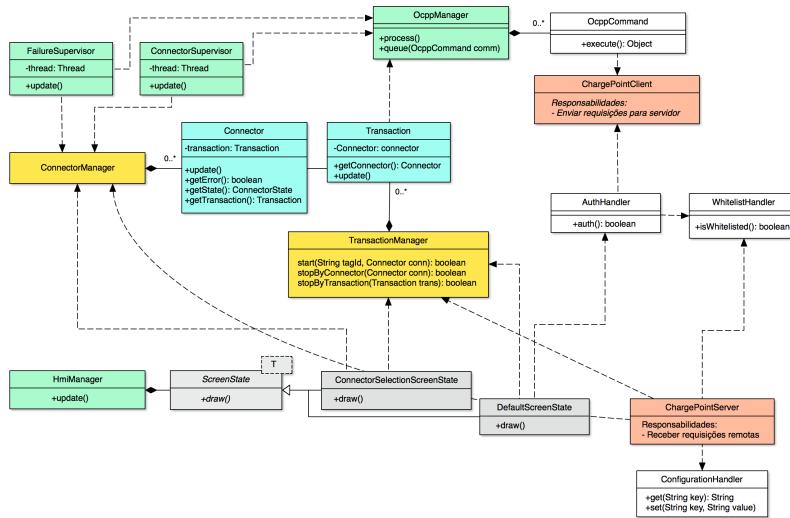


Figura 15 – Diagrama UML simplificado do projeto

3.4.2 Inicialização

A figura 16 apresenta como o programa é inicializado. Deve-se levar em conta que a primeira coisa a ser executada, antes mesmo da sequência definida no `main()`, é a inicialização do *Guice*, onde todos *check-ups* iniciais são feitos para verificar se não há nenhuma declaração errada em sua configuração.

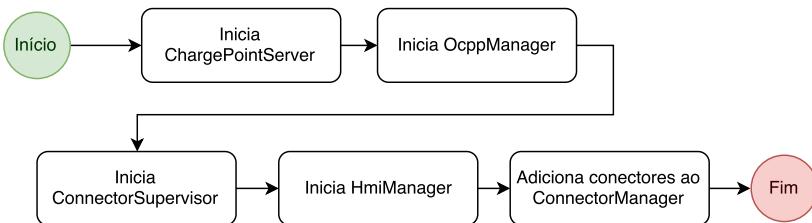


Figura 16 – Sequência de Inicialização

Todas classes que são instâncias independentes são inicializadas, menos a classe *Transaction* que só é inicializada quando há uma transação à ser iniciada. Visando prover agilidade, as configurações dos conectores são declaradas no corpo da classe que contém o *main()*, porém foi observado que poderia ser criada uma classe que adquirisse os dados da configuração do conector (atualmente em *settings.properties*) e adicionasse esses conectores ao *ConnectorManager*.

3.4.3 Inicialização e Finalização de Transações pelo Usuário

Para inicializar uma transação (carregamento), o usuário precisa ser registrado no sistema, sendo que este é validado por meio de uma *tag RFID*. A figura 17 demonstra o processo que o usuário deve executar para inicializar a transação com uma *tag RFID*.

Após encostar a *tag* no leitor, o *HmiManager* chama o *AuthHandler* para verificar se o usuário está ou não habilitado para carregar veículos na estação. Caso não, uma mensagem de erro é exibida e a IHM retorna para a tela inicial.

Após autenticar, o usuário precisa escolher um dos conectores disponíveis. Assim que escolhido, o *HmiManager* chama o *TransactionManager*, ao qual se encarrega de inicializar uma transação (*Transaction*). Ao inicializar a transação, uma *Thread* é criada. A primeira tarefa desta é habilitar o carregamento, chamando a classe *Conector* relacionada para tal. Por fim, o servidor OCPP é notificado da transação e o usuário recebe uma mensagem notificando que seu carregamento foi iniciado.

Também é possível inicializar uma transação por meio de um aplicativo. Nesse caso, o servidor OCPP mandará uma *RemoteStart-Transaction*, que iniciaria o mesmo processo como se fosse uma *tag RFID*, porém feito por meio de um servidor e não fisicamente.

A finalização de um carregamento é dada basicamente da mesma maneira e é apresentada na figura 18. Uma das diferenças é que, após o usuário ser autenticado, é checado se ele possui uma transação ativa. Caso houver, aparecerá o conector que ele pode cancelar o carregamento. Os outros processos são o inverso da inicialização (para a *Thread* e desabilita carregamento).

Um carregamento pode também ser finalizado pelo *Connector-Supervisor*, de forma automática, nos seguintes casos:

- Alguma falha durante o carregamento

- Conector desconectado do veículo por parte do usuário
- Carregamento ter chego ao fim (consumo próximo de zero)

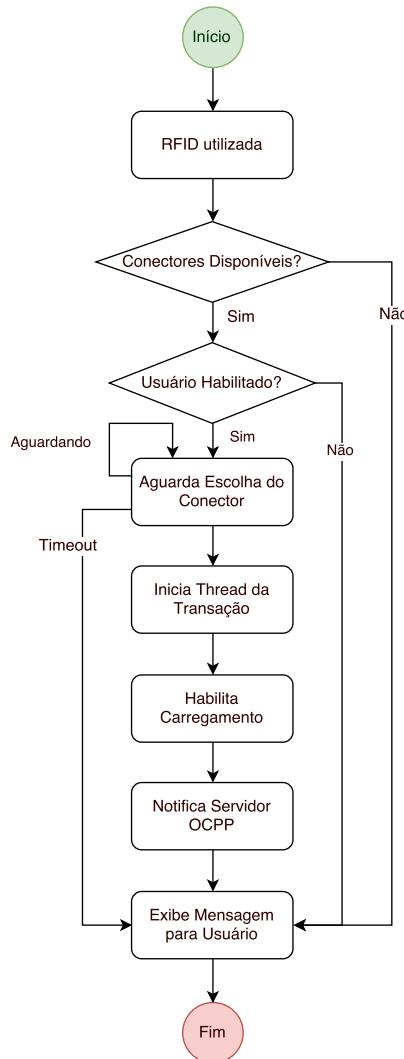


Figura 17 – Diagrama de Inicialização de uma Transação pelo usuário

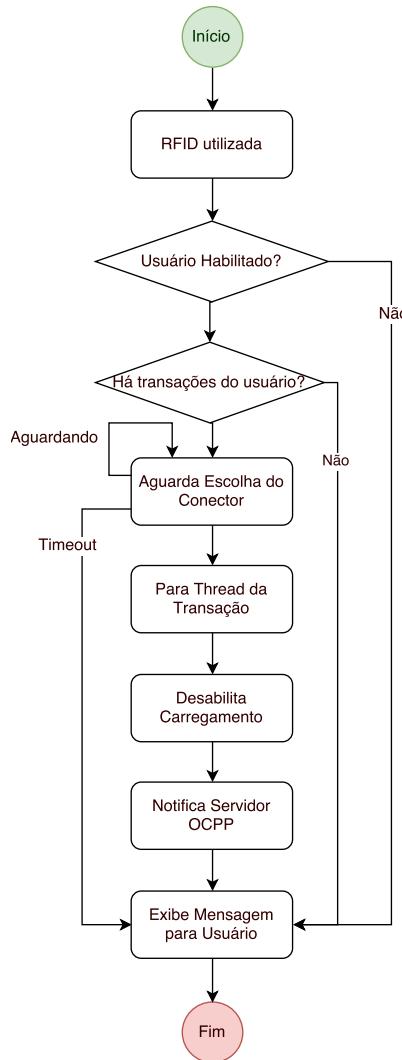


Figura 18 – Diagrama de Finalização de uma Transação pelo usuário

4 TESTES E RESULTADOS

4.1 FERRAMENTAS DE TESTE

Para testar a implementação realizada, foi necessária uma ferramenta para simular um servidor central. Inicialmente, testou-se a ferramenta GIR OCPPJS, porém essa apresentou problemas ao lidar com algumas das requisições vindas da *EVSE*. Entre esses problemas está o fato de ela não suportar reconexões (em caso do *software* da *EVSE* reiniciar) e, após certo período, a ferramenta não respondia às requisições.

Foi criada então uma ferramenta para testes, disponível em <https://github.com;brnluiz/ocpp-tools>. Essa ferramenta implementa somente o *OCPP 1.5 SOAP*, visto que a estação não suportaria *WebSocket*, com foco nas operações apresentadas na tabela 3.

Ao executá-la, todas requisições realizadas por estações conectadas são recebidas e a ferramenta responde com dados genéricos, o que possibilita testar o funcionamento da implementação do protocolo na *EVSE*. Caso o desenvolvedor precise realizar alguma requisição às estações, é possível utilizar a *Command Line Interface (CLI)* para tal. Apenas as requisições ReserveNow, GetDiagnostics, UpdateFirmware e SendLocalList não foram implementadas.

A figura 19 dá uma amostra da interface do programa. Como pode ser observado, a ferramenta dá ao usuário uma *CLI* onde ele pode inserir comandos que realizarão requisições *OCPP*. Na figura, o usuário realiza a requisição RemoteStartTransaction passando apenas remotestart como comando. Todos comandos são executados dessa maneira: colocando o nome do comando *OCPP* em conjunto com alguns parâmetros. Caso o usuário não souber quais parâmetros deve utilizar, ele pode somente digitar o nome do comando e então uma mensagem de ajuda será exibida. Além de enviar requisições, a ferramenta também exibe as requisições que foram enviadas pela estação, sendo nesse caso o BootNotification, StartTransaction e Heartbeat.

```

TrambiqMacBook:ocpp-0.1 brunoluz$ java -jar ocpp.jar
19:11:47 [INFO ] ocpp.CentralSystemClient5: Starting Central System OCPP client @ http://127.0.0.1:9000/ocpp/ChargePoint)
19:11:48 [INFO ] ocpp.CentralSystemClient5: Central System OCPP client has connected!
19:11:48 [INFO ] ocpp.CentralSystemServer15: Starting central system OCPP server @ http://0.0.0.0:9001/ocpp/CentralSystem

Mar 12, 2017 7:11:51 PM com.sun.xml.ws.server.MonitorBase createRoot
INFO: Metro monitoring rootname successfully set to: com.sun.metro:pp=/,type=WSEndpoint,name=CentralSystemServer15Service-CentralSystemServer15Port
19:11:51 [INFO ] ocpp.CentralSystemServer15: Central system OCPP server is published!
19:11:51 [INFO ] chargesystem.UserInputManager: Starting UserInputManager...
ocpp@cs:cp# 19:11:51 [INFO ] ocpp.CommandsDispatcher: Starting CommandsDispatcher...
ocpp@cs:cp# 19:11:56 [INFO ] ocpp.CentralSystemServer15: Received ocpp.cs._2012._06.BootNotificationRequest

ocpp@cs:cp# remotestart
usage: RemoteStartTransactionCommand
  -c,<--connector ><arg>    connector id
  -cpid,<--chargepoint ><arg>  charge point id
  -t,<--tag ><arg>           tagid
19:12:02 [WARN ] chargesystem.UserInputManager: Command not executed
ocpp@cs:cp# remote -c 1 -cpid 1 -t 1234
19:12:08 [WARN ] chargesystem.UserInputManager: Invalid command or not implemented
ocpp@cs:cp# remotestart -c 1 -cpid 1 -t 1234
ocpp@cs:cp# 19:12:18 [INFO ] ocpp.CentralSystemClient5: Sending ocpp.cp._2012._06.RemoteStartTransactionRequest
19:12:18 [INFO ] ocpp.cp.commands.ChangeAvailabilityCommand: Received status: ACCEPTED
19:12:19 [INFO ] ocpp.CentralSystemServer15: Received ocpp.cs._2012._06.RemoteStartTransactionRequest
19:12:19 [INFO ] ocpp.CentralSystemServer15: CentralSystemServer15: transaction id = 1489356739
ocpp@cs:cp# 19:12:28 [INFO ] ocpp.CentralSystemServer15: Received ocpp.cs._2012._06.HeartbeatRequest

```

Figura 19 – Ferramenta que simula servidor OCPP

4.2 IMPLEMENTAÇÕES E TESTES INICIAIS

O desenvolvimento inicial foi realizado a partir de código fonte existente, no qual não havia implementação funcional da EVSE e apenas possibilitava o teste de cada dispositivo isoladamente. Após essa etapa de revisão de código, iniciou-se a implementação do software da estação.

Alguns testes unitários foram desenvolvidos sob o *framework* de testes *JUnit*, o que permite o programa ser testado antes de ser embarcado. Após concluir com sucesso a etapa de verificação do software, ou seja, quando os testes unitários não detectam erros, o programa é enviado para a *BeagleBone*. Conforme apresentado na Figura 20, a BeagleBone está conectada aos dispositivos listados na seção 3.1, porém dispostos em uma bancada de testes.

Como as cargas utilizadas durante o teste consumiam pouca energia (fontes CA-CC de dispositivos de baixo consumo, como notebooks), o medidor de energia teve seu multiplicador de corrente configurado em 100, possibilitando assim simular um consumo alto de forma rápida (sem precisar aguardar um grande período de tempo para atingir 1 kWh).

Simulou-se a inicialização e finalização de cargas no conector modo 2, onde foram seguidos fluxos similares as figuras 17 e 18, já que essas permitem verificar diversos aspectos do sistema: autenticação, co-

municação externa e com medidores, criação de transações, verificação de falhas, entre outras.

O conector modo 3 Mennekes não pode ser testado na bancada de testes pois, para habilitar o carregamento, é necessário realizar todo sequenciamento apresentado na figura 1 e este só seria possível se houvesse um veículo próximo a bancada. Portanto, este ficou para ser testado na própria estação protótipo.

Todos arquivos, incluindo o sistema operacional, são colocados em um cartão SD, o que facilita a cópia desses dados para a realização de cópias de segurança (criação de imagens), possibilitando o intercâmbio do sistema entre a bancada de testes e a estação protótipo.

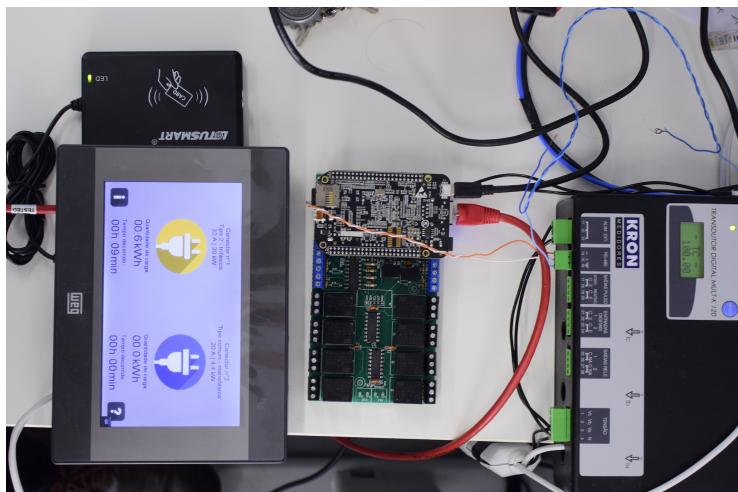


Figura 20 – Disposição dos dispositivos na bancada de testes

4.3 TESTES NA ESTAÇÃO PROTÓTIPO

Após executar os testes na bancada, o sistema foi levado para a EVSE protótipo, situada no estacionamento da Fundação CERTI (figura 22). O arranjo dos dispositivos da estação difere da bancada de testes (figura 21), porém visto que os dispositivos são os mesmos e o sistema embarcado foi colocado em um cartão SD, só seria necessário inserir o cartão na BeagleBone da estação e configurá-la para realizar a inicialização a partir do cartão, caso já não estiver configurada de

fábrica.

Visto que a estação, no momento da escrita deste projeto, se encontrava sem conectividade, foi utilizado o modo local. Nesse modo, o *software* captura as chamadas que seriam enviadas para o *Central System* e as responde localmente, permitindo seu funcionamento mesmo sem conectividade.



Figura 21 – Disposição dos dispositivos na estação protótipo

Embora a previsão fosse de que a estação apresentasse o mesmo comportamento da bancada de testes, ao menos para o conector modo 2, não foi isso que ocorreu inicialmente. Um dos problemas observados foi relativo a conectividade da BeagleBone com outros dispositivos Ethernet, que foi resolvido com a substituição do *connman* pelo *Network Manager*. O *connman* possui configurações atreladas a dados únicos de cada dispositivo, o que necessitaria que cada BeagleBone fosse configurada manualmente. Enquanto isso, o *Network Manager* se baseia em configurações fixas no arquivo `/etc/network/interfaces`, sendo que esse não causou problemas quando executado na estação.

Outros problemas observados estavam relacionados à comuni-

cação serial: um foi devido as configurações seriais da bancada e da estação serem diferentes, porém isso foi rapidamente detectado e resolvido. Porém, um problema um pouco mais complicado surgiu logo após, quando notamos que o *software* não estava funcionando adequadamente na estação pois não conectava ou perdia leituras. Foi descoberto que alguns circuitos integrados que transformam a saída *UART* para RS-485 possuem sua saída conectada a sua entrada, o que acabava gerando problemas na recepção de alguns dados. Após alguns dias foi descoberto que isso se chama *echo* e poderia ser resolvido via *software* (parâmetro "*echo*" da classe de conexão do Jamod).



Figura 22 – Estação protótipo final com Mitsubishi Miev estacionado

Como não foi possível testar o conector modo 3 *Mennekes* previamente, alguns problemas surgiram também. O conector do veículo de testes, Mitsubishi Miev, usa o padrão americano. Portanto, é necessário um adaptador para o padrão *Mennekes* para carregá-lo. Porém, tal adaptador gerou falhas elétricas que impossibilitavam o carregamento contínuo do veículo, como mal contato entre os pinos do adaptador e do conector, visto que o *Phoenix* desativa o carregamento imediatamente ao perceber alguma falha. Para resolver esse problema, o adaptador foi cortado em alguns milímetros para possibilitar que os pinos de contato ficassem mais próximos, além de o *software* mandar um comando para o *Phoenix* limitar a corrente máxima permitida (visto que há suspeita de que, devido ao adaptador, o carregamento não pode exigir muita

potência). Deste modo foi possível fazer alguns carregamentos experimentais, porém nenhum longo foi realizado devido a limitações de tempo e problemas com o veículo.

Em um teste anterior com um BMW i3 sem o *software* de controle, foi realizado um carregamento longo, sem nenhuma falha, com o conector *Mennekes*. A figura 23 mostra a curva de corrente x tempo do carregamento, onde a estação chega a fornecer um máximo de 25,1 A e levou aproximadamente 4h30m para finalizar o carregamento. Poderia ser ainda mais rápido caso a estação possuísse um conector modo 4 (seção 2.2.1.4).

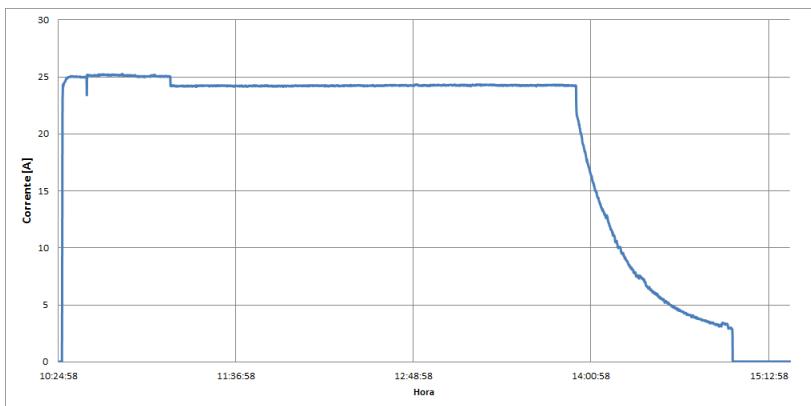


Figura 23 – Gráfico Corrente x Tempo para carregamento do BMW i3 utilizando o conector modo 3 *Mennekes*

Foi realizada uma carga de menos de um minuto apenas para mostrar o sequenciamento da estação, visto que o programa gera saídas de texto para diversas operações realizadas internamente e isso permite a visualização do que está ocorrendo na estação. A saída completa está disponível no apêndice A. Nos trechos abaixo, é possível observar o *software* realizar algumas das etapas mostradas nos sequenciamentos das figuras 16, 17 e 18.

- Linha 3-23: inicialização do programa

```

1 19:22:41 [INFO ] app.App: Start of main program sequence!
2 19:23:02 [INFO ] auth.WhitelistHandler: Whitelist has 8 entries. (version=1)
3 Initialized pin P9.12 with direction OUT.
4 Set GPIO P9.12 signal to LOW

```

```

5 | RXTX Warning: Removing stale lock file. /var/lock/LCK..ttyS1
6 | 19:23:03 [INFO ] energy.EnergyCalculator: Starting EnergyCalculator and requester
...
7 | 19:23:03 [INFO ] energy.EnergyCalculator: EnergyCalculator started!
8 | 19:23:04 [INFO ] connectors.CommonConnector: Kron Series number: 1868664
9 | 19:23:04 [INFO ] connectors.ConnectorManager: Added one connector (PLUG_COMMON) !
Actual size: 1
10 | 19:23:05 [INFO ] connectors.PhoenixControllerConnector: Kron Series number:
1857627
11 | 19:23:05 [INFO ] connectors.ConnectorManager: Added one connector (
CONTROLLER_TYPE2PHOENIX) ! Actual size: 2
12 | 19:23:05 [INFO ] connectors.ConnectorsSupervisor: Starting supervisor...
13 | 19:23:05 [INFO ] ocpp.OcppManager: Starting OcppManager...
14 | 19:23:05 [INFO ] connectors.ConnectorsSupervisor: Supervisor started!
15 | 19:23:05 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
transaction connectors
16 | 19:23:05 [INFO ] ocpp.OcppManager: OcppManager started
17 | 19:23:05 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
BootNotificationRequest
18 | 19:23:05 [INFO ] failures.FailureSupervisor: Starting failure supervisor...
19 | 19:23:05 [INFO ] failures.FailureSupervisor: Failure supervisor started!
20 | 19:23:06 [INFO ] hmi.HmiManager: Starting Hmi...
21 | 19:23:06 [INFO ] hmi.HmiManager: Hmi Started!
```

- Linha 34: usuário utiliza a *RFID*

```
1 | 19:23:22 [DEBUG] hmi.WegHmiDevice: Got RFID tag: 4D16DC27
```

- Linha 35-39: *RFID* é autenticada (usuário habilitado para carregamento)

```

1 | 19:23:22 [INFO ] auth.AuthHandler: Tag 4D16DC27 is whitelisted!
2 | 19:23:22 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from LOADING to
AUTH_SUCCESSFUL
3 | 19:23:22 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
4 | 19:23:23 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
5 | 19:23:24 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from AUTH_SUCCESSFUL to
CONNECTOR_SELECTION
```

- Linhas 39-41: *software* aguarda usuário selecionar o conector

```

1 | 19:23:24 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from AUTH_SUCCESSFUL to
CONNECTOR_SELECTION
2 | 19:23:25 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
transaction connectors
3 | 19:23:26 [INFO ] transactions.TransactionManager: Starting charge via OCPP on
connector 0. User ID tag: 4D16DC27
```

- Linhas 42-45: inicia a *Thread* da transação e habilita o carregamento

```

1 | 19:23:26 [INFO ] transactions.TransactionImpl: Starting Transaction...
2 | 19:23:26 [INFO ] transactions.TransactionImpl: Connector 0 - Starting charging
operation #
3 | Set GPIO P9.12 signal to HIGH
4 | 19:23:26 [INFO ] transactions.TransactionImpl: Transaction Started!
```

- Linha 46: mostra mensagem para o usuário sobre o início do carregamento

```
1 19:23:27 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from CONNECTOR_SELECTION
to CHARGING_STARTED
```

- Linhas 48-54: notifica servidor *OCPP* sobre o início do carregamento

```
1 19:23:28 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
StatusNotificationRequest
2 19:23:28 [INFO ] ocpp.LocalChargePointClient15: Connector: 0
3 19:23:28 [INFO ] ocpp.LocalChargePointClient15: - Status: OCCUPIED
4 19:23:28 [INFO ] ocpp.LocalChargePointClient15: - Error code: NO_ERROR
5 19:23:28 [INFO ] ocpp.LocalChargePointClient15: - Info:
6 19:23:28 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
StartTransactionRequest
7 19:23:28 [INFO ] ocpp.LocalChargePointClient15: LocalChargePointClient15:
transaction id = 1486409008
```

- Linhas 55-86: *software* envia dados para servidor e alterna entre algumas telas na IHM

```
1 19:23:29 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from CHARGING_STARTED to
DEFAULT
2 19:23:30 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
transaction connectors
3 19:23:33 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
4 19:23:33 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
0
5 19:23:33 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
6 19:23:35 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
transaction connectors
7 19:23:38 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
8 19:23:38 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
0
9 19:23:38 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
10 19:23:39 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from DEFAULT to
AUTH_SCREEN
11 19:23:41 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
transaction connectors
12 19:23:41 [DEBUG] transactions.TransactionImpl: Transaction 1486409008: (charging
time: 15)
13 19:23:43 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
14 19:23:43 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
0
15 19:23:43 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
0
16 19:23:43 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
17 19:23:46 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
transaction connectors
18 19:23:46 [DEBUG] transactions.TransactionImpl: Transaction 1486409008: (charging
time: 20)
19 19:23:48 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
20 19:23:48 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
0
21 19:23:48 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
0
```

```

22 | 19:23:48 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
23 | 19:23:49 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from AUTH_SCREEN to
24 | DEFAULT
24 | 19:23:51 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
25 | transaction connectors
25 | 19:23:51 [DEBUG] transactions.TransactionImpl: Transaction 1486409008: (charging
26 | time: 25)
26 | 19:24:14 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
27 | HeartbeatRequest
27 | 19:24:14 [DEBUG] utils.SystemUtils: SYNCHRONIZING CLOCK: setting date to
27 | 2017-02-06 19:24:14
28 | 19:24:24 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
29 | 19:24:24 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
29 | 0
30 | 19:24:24 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
31 | 19:24:28 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
31 | transaction connectors

```

- Linhas 87-88: carregamento cancelado devido desconexão do veículo por parte do usuário

```

1 | 19:24:28 [INFO ] transactions.TransactionImpl: Connector 0 - Ending charging
1 | operation #1486409008
2 | Set GPIO P9.12 signal to LOW

```

- Linhas 89-95: notifica servidor *OCPP* sobre o término do carregamento

```

1 | 19:24:29 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
2 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
2 | StatusNotificationRequest
3 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: Connector: 0
4 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: - Status: AVAILABLE
5 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: - Error code: NO_ERROR
6 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: - Info:
7 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
7 | StopTransactionRequest

```


5 CONCLUSÃO

O objetivo de estudar o funcionamento de uma *EVSE* e como implementá-la foi alcançado com sucesso após os testes finais na estação protótipo.

O *software* desenvolvido nesse trabalho acadêmico é um dos elementos fundamentais para todo o projeto da estação. A estação em si é um produto pioneiro no Brasil, visto que essa é a primeira a ser desenvolvida em solo nacional (não há nenhum produto semelhante, desenvolvido por outras empresas nacionais), visando testes e análises de viabilidade para futuras oportunidades.

A complexidade do *software*, assim como todas tecnologias utilizadas, permitiram que os objetivos relacionados à *software* fossem alcançados também. Embora houveram problemas ao transferir a aplicação da bancada de testes para a estação, estes permitiram o aprendizado de métodos para resolução de problemas.

Durante o projeto, alguns dos desafios encontrados foram relacionados a camadas de software entre o sistema operacional e o hardware. Embora tenham desacelerado um pouco o projeto, esses desafios permitiram o desenvolvimento de um conhecimento de Linux embarcado, que possui algumas peculiaridades quando comparado a versão *desktop*, como *Device Trees*, gerenciador de boot diferenciado (*GRUB* x *uBoot*) e acesso facilitado a recursos da placa utilizada (periféricos, entradas/-saídas e serial).

Uma das sugestões de modificação do projeto é a utilização de medidores integrados à BeagleBone por meio das entradas analógicas ou digitais (caso a saída do medidor for codificada). Isso facilitaria o desenvolvimento futuro de uma placa mais otimizada, com todos dispositivos integrados à mesma (micro-controlador, comunicação serial, acionamentos e entradas).

O mercado de veículos elétricos se mostra uma área promissora para qualquer engenheiro eletricista, visto que apresenta desafios e oportunidades em diferentes competências. Embora o foco desse projeto tenha sido a implementação do protocolo e do *software* de controle, foi possível não só aprender sobre isso, mas também sobre outros aspectos desse mercado, visto que no ambiente de trabalho houveram diversas conversas sobre assuntos relacionados (regulamentação, equipamentos, novas tecnologias, tendências e entre outros).

5.1 RECONHECIMENTO

Os trabalhos descritos neste documento foram realizados no âmbito do projeto "Sistema de Recarga Rápida com Armazenamento Híbrido-Estacionário de Energia para Abastecimento de Veículos Elétricos no Conceito de Redes Inteligentes"(P&D ANEEL 5697-0414/2014), financiado pelo programa de P&D da ANEEL, viabilizado pela CELESC Distribuição S.A. e executado pela Fundação CERTI.

Referências

ABNT. NBR 14136 Plugues e tomadas para uso doméstico e análogo até 20 A/250 V em corrente alternada - Padronização: 2012. 2012.

BLOCK, D.; HARRISON, J.; BROOKER, P. **Electric Vehicle Sales for 2014 and Future Projections**. 2015. Disponível em: <<http://www.fsec.ucf.edu/en/publications/pdf/FSEC-CR-1998-15.pdf>>. Acesso em: 30/12/2016.

BUAMOD, I.; ABDELMOGHITH, E.; MOUFTAH, H. T. A review of osi-based charging standards and emobility open protocols. In: **2015 6th International Conference on the Network of the Future (NOF)**. 2015. p. 1-7.

CAAM. **New energy vehicles enjoyed a high-speed growth**. 2016. Disponível em: <<http://www.caam.org.cn-AutomotivesStatistics/20160120/1305184260.html>>. Acesso em: 30/12/2016.

CHRISTENSEN, E. et al. **Web Services Description Language (WSDL) 1.1**. 2001.
[Https://www.w3.org/TR/2001/NOTE-wsdl-20010315](https://www.w3.org/TR/2001/NOTE-wsdl-20010315).

DEVELOPER.COM. **Automated Unit Testing Frameworks**. 2004. Disponível em: <<http://www.developer.com/java/ent/article.php/3372151/Automated-Unit-Testing-Frameworks.htm>>. Acesso em: 18/01/2017.

EAA. **Electric Vehicles in Europe**. 2016. Disponível em: <<http://www.eea.europa.eu/publications/electric-vehicles-in-europe-download>>. Acesso em: 30/12/2016.

FOX, G. H. Electric vehicle charging stations: Are we prepared? **IEEE Industry Applications Magazine**, v. 19, n. 4, p. 32-38, July 2013. ISSN 1077-2618.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-oriented Software**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

GUDGIN, M. et al. **SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)**. apr 2007.

[Http://www.w3.org/TR/2007/REC-soap12-part1-20070427/](http://www.w3.org/TR/2007/REC-soap12-part1-20070427/).

IEC. **International Standard IEC 62196-3**. 2014.

KELLY-DETWILER, P. **Building Out The Electric Vehicle Charging Infrastructure: Greenlots Advocates For Open Standards**. 2014. Disponível em: <<http://www.forbes.com/sites-peterdetwiler/2014/03/13/building-out-the-electric-vehicle-charging-infrastructure-greenlots-advocates-for-open-standards>>. Acesso em: 29/12/2016.

LONGO, M. et al. Recharge stations: A review. In: **2016 Eleventh International Conference on Ecological Vehicles and Renewable Energies (EVER)**. 2016. p. 1–8.

MACHINEDESIGN. **Could Battery Swapping Ease Range Anxiety for EV Owners?** 2016. Disponível em: <<http://machinedesign.com/automotive/could-battery-swapping-ease-range-anxiety-ev-owners>>. Acesso em: 30/01/2016.

MCKINSEY. **Electric vehicles in Europe: gearing up for a new phase?** 2014. Disponível em: <<http://www.w3.org/TR/2001/NOTE-wsdl-20010315>>. Acesso em: 04/12/2016.

MENNEKES. **Type 2 charging plug proposed as the common standard for Europe**. 2013. Disponível em: <http://www.mennekes.de/uploads/media/MENNEKES_Media_information-Type_2_charging_plug_proposed_as_the_common_standard_for_Europe.pdf>. Acesso em: 04/01/2017.

MODBUS. **MODBUS Application Protocol Specification v1.1b3**. 2012. Disponível em: <http://www.modbus.org/docs-Modbus_Application_Protocol_V1_1b3.pdf>. Acesso em: 29/12/2016.

OCA. **OCPP 1.5 specification**. 2012. Disponível em: <http://www.openchargealliance.org/uploads/files/protected-ocpp_specification_1.5_final.pdf>. Acesso em: 05/10/2016.

OCA. Standardization of OCPP. 2016. Disponível em: <<http://www.openchargealliance.org/oasis/>>. Acesso em: 30/12/2016.

PHOENIXCONTACT. PHOENIX CONTACT | AC charging controller - EM-CP-PP-ETH - 2902802. 2016. Disponível em: <<https://www.phoenixcontact.com/online/portal/us?uri=pxc-oc-itemdetail:pid=2902802>>. Acesso em: 30/01/2016.

SCHMITT, B. Germany's Bundesrat Resolves End Of Internal Combustion Engine. 2016. Disponível em: <<http://www.forbes.com/sites/bertelschmitt/2016/10/08/germany-s-bundesrat-resolves-end-of-internal-combustion-engine>>. Acesso em: 29/12/2016.

TESLA. Supercharger | Tesla. 2016. Disponível em: <<https://www.tesla.com/supercharger>>. Acesso em: 30/01/2016.

THESERVERSIDE. Spring vs. Guice: The Clash of the IOC Containers. 2010. Disponível em: <<http://www.theserverside.com-feature/Spring-vs-Guice-The-Clash-of-the-IOC-Containers>>. Acesso em: 18/01/2017.

Apêndice A – Log de Sequenciamento de Carregamento


```
1 sputnik@beaglebone:~$ /opt/evse/run.sh
2 Listening for transport dt_socket at address: 6006
3 19:22:41 [INFO ] app.App: Start of main program sequence!
4 19:23:02 [INFO ] auth.WhitelistHandler: Whitelist has 8 entries. (version=1)
5 Initialized pin P9.12 with direction OUT.
6 Set GPIO P9.12 signal to LOW
7 RXTX Warning: Removing stale lock file. /var/lock/LCK..ttyS1
8 19:23:03 [INFO ] energy.EnergyCalculator: Starting EnergyCalculator and requester
...
9 19:23:03 [INFO ] energy.EnergyCalculator: EnergyCalculator started!
10 19:23:04 [INFO ] connectors.CommonConnector: Kron Series number: 1868664
11 19:23:04 [INFO ] connectors.ConnectorManager: Added one connector (PLUG_COMMON) !
Actual size: 1
12 19:23:05 [INFO ] connectors.PhoenixControllerConnector: Kron Series number:
1857627
13 19:23:05 [INFO ] connectors.ConnectorManager: Added one connector (
CONTROLLER_TYPE2PHOENIX)! Actual size: 2
14 19:23:05 [INFO ] connectors.ConnectorsSupervisor: Starting supervisor...
15 19:23:05 [INFO ] ocpp.OcppManager: Starting OcppManager...
16 19:23:05 [INFO ] connectors.ConnectorsSupervisor: Supervisor started!
17 19:23:05 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
transaction connectors
18 19:23:05 [INFO ] ocpp.OcppManager: OcppManager started
19 19:23:05 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
BootNotificationRequest
20 19:23:05 [INFO ] failures.FailureSupervisor: Starting failure supervisor...
21 19:23:05 [INFO ] failures.FailureSupervisor: Failure supervisor started!
22 19:23:06 [INFO ] hmi.HmiManager: Starting Hmi...
23 19:23:06 [INFO ] hmi.HmiManager: Hmi Started!
24 19:23:07 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from BOOT to DEFAULT
25 19:23:07 [DEBUG] utils.SystemUtils: SYNCHRONIZING CLOCK: setting date to
2017-02-06 19:23:05
26 19:23:10 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
transaction connectors
27 19:23:12 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
28 19:23:12 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
29 19:23:15 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
transaction connectors
30 19:23:17 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from DEFAULT to
AUTH_SCREEN
31 19:23:17 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
32 19:23:17 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
33 19:23:20 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
transaction connectors
34 19:23:22 [DEBUG] hmi.WegHmiDevice: Got RFID tag: 4D16DC27
35 19:23:22 [INFO ] auth.AuthHandler: Tag 4D16DC27 is whitelisted!
36 19:23:22 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from LOADING to
AUTH_SUCCESSFUL
37 19:23:22 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
38 19:23:23 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
39 19:23:24 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from AUTH_SUCCESSFUL to
CONNECTOR_SELECTION
40 19:23:25 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
transaction connectors
41 19:23:26 [INFO ] transactions.TransactionManager: Starting charge via OCPP on
connector 0. User ID tag: 4D16DC27
42 19:23:26 [INFO ] transactions.TransactionImpl: Starting Transaction...
43 19:23:26 [INFO ] transactions.TransactionImpl: Connector 0 - Starting charging
operation #
44 Set GPIO P9.12 signal to HIGH
45 19:23:26 [INFO ] transactions.TransactionImpl: Transaction Started!
```

```
46 | 19:23:27 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from CONNECTOR_SELECTION
47 | to CHARGING_STARTED
48 | 19:23:28 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
49 | 19:23:28 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
50 | StatusNotificationRequest
51 | 19:23:28 [INFO ] ocpp.LocalChargePointClient15: Connector: 0
52 | 19:23:28 [INFO ] ocpp.LocalChargePointClient15: - Status: OCCUPIED
53 | 19:23:28 [INFO ] ocpp.LocalChargePointClient15: - Error code: NO_ERROR
54 | 19:23:28 [INFO ] ocpp.LocalChargePointClient15: - Info:
55 | 19:23:28 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
56 | StartTransactionRequest
57 | 19:23:28 [INFO ] ocpp.LocalChargePointClient15: LocalChargePointClient15:
58 | transaction id = 1486409008
59 | 19:23:28 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
60 | 19:23:29 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from CHARGING_STARTED to
61 | DEFAULT
62 | 19:23:30 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
63 | transaction connectors
64 | 19:23:33 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
65 | 19:23:33 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
66 | 0
67 | 19:23:33 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
68 | 19:23:35 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
69 | transaction connectors
70 | 19:23:38 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
71 | 19:23:38 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
72 | 0
73 | 19:23:38 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
74 | 19:23:39 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from DEFAULT to
75 | AUTH_SCREEN
76 | 19:23:41 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
77 | transaction connectors
78 | 19:23:41 [DEBUG] transactions.TransactionImpl: Transaction 1486409008: (charging
79 | time: 15)
80 | 19:23:43 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
81 | 19:23:43 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
82 | 0
83 | 19:23:43 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
84 | 0
85 | 19:23:43 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
86 | 19:23:46 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
87 | transaction connectors
88 | 19:23:46 [DEBUG] transactions.TransactionImpl: Transaction 1486409008: (charging
89 | time: 20)
90 | 19:23:48 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
91 | 19:23:48 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
92 | 0
93 | 19:23:48 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
94 | 0
95 | 19:23:48 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
96 | 19:23:49 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from AUTH_SCREEN to
97 | DEFAULT
98 | 19:23:51 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
99 | transaction connectors
100 | 19:23:51 [DEBUG] transactions.TransactionImpl: Transaction 1486409008: (charging
101 | time: 25)
102 | 19:24:14 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
103 | HeartbeatRequest
104 | 19:24:14 [DEBUG] utils.SystemUtils: SYNCHRONIZING CLOCK: setting date to
105 | 2017-02-06 19:24:14
106 | 19:24:24 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
```

```
84 | 19:24:24 [DEBUG] connectors.AbstractConnector: Sending meter values for connector
85 | 0
85 | 19:24:24 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
86 | 19:24:28 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
     transaction connectors
87 | 19:24:28 [INFO ] transactions.TransactionImpl: Connector 0 - Ending charging
     operation #1486409008
88 | Set GPIO P9.12 signal to LOW
89 | 19:24:29 [DEBUG] ocpp.OcppManager: Starting to process the OCPP Commands Queue
90 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
     StatusNotificationRequest
91 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: Connector: 0
92 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: - Status: AVAILABLE
93 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: - Error code: NO_ERROR
94 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: - Info:
95 | 19:24:29 [INFO ] ocpp.LocalChargePointClient15: Received ocpp.cs._2012._06.
     StopTransactionRequest
96 | 19:24:29 [DEBUG] ocpp.OcppManager: Finished OCPP Commands Queue processing
97 | 19:24:30 [DEBUG] hmi.HmiManager: Hmi ScreenState changed from AUTH_SCREEN to
     DEFAULT
98 | 19:24:33 [DEBUG] connectors.ConnectorsSupervisor: Reading measurements from all
     transaction connectors
```